Universidade do Minho, Departamento de Informática
MIEI, 2020-2021

Comunicação de Dados

Trabalho Prático Nº1

v4.2

Implementar um programa em C que, mediante um ficheiro de entrada, gere um ficheiro comprimido obtido por codificação Shannon-Fano (SF). Não se trata de codificação Huffman!

Os símbolos a considerar da fonte/ficheiro são bytes individuais (ou bloco de oito bits). A ferramenta também deve implementar a funcionalidade complementar de descomprimir um ficheiro previamente comprimido.

O código deve utilizar apenas funções normalizadas da linguagem C (norma 217 ISSO - STD 17) e/ou funções desenvolvidas pelo grupo de trabalho. Não use APIs, funções ou excertos de código de terceiros. Não podem ser usadas APIs externas para construção do código. É aconselhado o uso de um dos seguintes ambientes IDE: Code::Blocks (recomendado), Eclipse, Visual Studio Code, GNAT ou NetBeans. De notar que estes IDEs não são compiladores e podem incluir ou não compiladores na sua distribuição. Os que não tiverem compiladores incluídos obrigarão à inclusão/instalação dum para poderem funcionar. Os docentes irão compilar o código entregue pelos alunos no IDE Code::Blocks com o compilador gcc/mingw incluído. Os alunos devem garantir que o código compila sem erros neste IDE, independente da plataforma (Windows 10, Linux, Mac OS).

O trabalho é de realização em grupos de 6 a 8 elementos (os elementos não precisam ser todos do mesmo turno TP). Os elementos dos grupos precisam fazer a sua inscrição no BB da UC a partir do dia 1 de dezembro. O relatório e o código devem ser entregues até às 23:59 do dia 3 de janeiro. As defesas dos trabalhos serão realizadas nos dias 5, 6 e 7 de janeiro em sessões *online* de duração aproximada de 60 minutos, em horário a combinar com os docentes.

A ferramenta a desenvolver deve ser constituída por quatro módulos independentes. Cada módulo será construído numa determinada fase de desenvolvimento e deve implementar um conjunto de funcionalidades ou requisitos especificados neste enunciado. Recomenda-se que o grupo seja dividido em subgrupos de dois elementos e que cada subgrupo seja responsável pela implementação dum módulo duma fase específica, pela escrita da secção do relatório dedicada à implementação dessa fase e à defesa desse módulo do programa. Todos os elementos serão responsáveis pela agregação dos módulos num único programa.

A avaliação dos módulos construídos terá em consideração os seguintes parâmetros: correção do código e das funcionalidades criadas; correção da sintaxe dos ficheiros criados; qualidade do código gerado; desempenho do código gerado; qualidade do relatório produzido e qualidade da defesa do trabalho realizado.

Recomenda-se que o código seja o mais legível possível, esteja o melhor documentado/comentado possível e inclua em todos os ficheiros um cabeçalho com a autoria do código, a data de criação/alteração, uma descrição das

principais funções criadas e variáveis utilizadas. Também se recomenda o uso de ficheiros do tipo **.h** para definição de tipos de variáveis e funções comuns a todos os módulos.

A secção "Otimização por execução paralela" deste enunciado apresenta recomendações para a escrita do relatório.

Introdução

A ferramenta deve chamar-se **shafa** e deve ser construída por módulos, em que cada módulo implementa uma funcionalidade (ou fase) específica, tem um determinado grupo de ficheiros de entrada e gera um determinado grupo de ficheiros de saída e um conjunto de mensagens textuais na consola. Em geral, os ficheiros de saída devem ficar com o mesmo nome dos ficheiros de entrada acrescentando uma terminação que identifica uma determinada operação. Cada módulo é identificado por um argumento do programa.

A Figura 1 apresenta a arquitetura recomendada para a ferramenta. De notar que existe um módulo adicional não representado na imagem e que apenas deve implementar um texto de ajuda/manual.

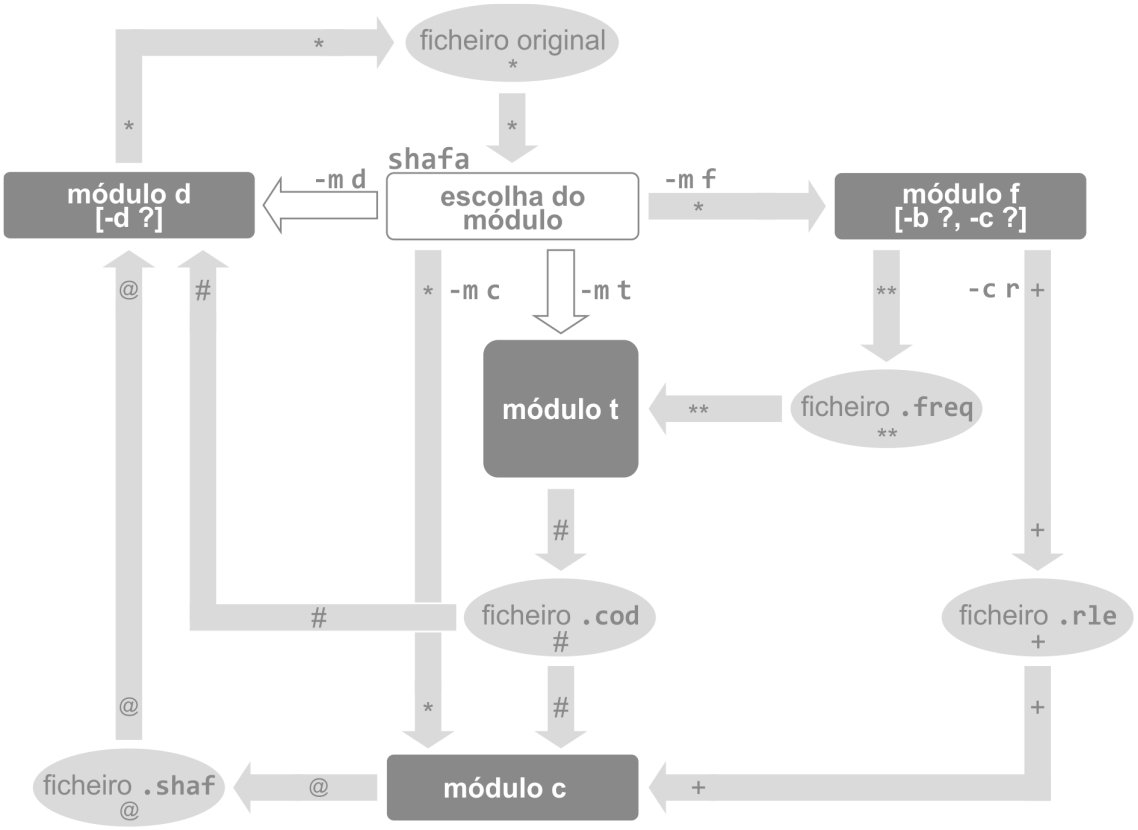

Figura 1: Arquitetura da aplicação **shafa**.

Nas secções seguintes é apresentada a especificação dos módulos a implementar no programa, bem como a descrição dos argumentos que o identificam, as opções possíveis e a sintaxe dos ficheiros de entrada e de saída esperados.

NOTA: Durante este enunciado serão usadas expressões especiais para representar a sintaxe do conteúdo dos ficheiros de saída resultantes da execução dos vários módulos do programa. Segue-se a sua explicação:

Sintaxe	*Semântica & Codificação em ficheiro*
$(b_1b_2b_3…b_N)$	Uma sequência de N bits guardado explicitamente; por exemplo, **(00001100)** representa o valor 12 guardado num byte (oito bits)
$\{V\}_N$	Um valor inteiro positivo V (V≥0), guardado explicitamente, ocupando N bytes; por exemplo, $\{0\}_1$ representa um byte com o valor 0 (em binário seria **00000000**)
[V]	Representa o valor do número inteiro positivo V (V≥0), guardado implicitamente como uma sequência de dígitos decimais, cada dígito guardado como um código ASCII de um byte; quando for óbvio nos exemplos deste enunciado, os parênteses podem não existir; por exemplo, **[13]** (ou apenas **13**) representa o valor 13 guardado como uma sequência de dois códigos ASCII, o do carater 1 seguido do código ASCII do carater 3; num ficheiro de saída, **[13]** é equivalente aos dois bytes seguintes: $\{49\}_1\{51\}_1$
[…]	Um ou mais valores do tipo anterior.
<S>	Representa a *string* de símbolos S guardada implicitamente como uma sequência de bytes com o código ASCII do carater respetivo; quando for óbvio nos exemplos deste enunciado, os parênteses podem não existir; por exemplo, **[ab]** (ou apenas **ab**) representa a sequência de dois códigos ASCII, o do carater **a** seguido do código ASCII do carater **b**; num ficheiro de saída, **[ab]** é equivalente aos dois bytes seguintes: $\{97\}_1\{98\}_1$
<…>	Um ou mais valores do tipo anterior.
$<S_1\|S_2\|…\|S_N>$	Um símbolo, de entre um grupo de símbolos possíveis, guardado como um código ASCII dum byte; ; nenhum dos símbolos pode ser um dos símbolos especiais **<@>** ou **<;>**; por exemplo, **<R|N>** representa que o byte do ficheiro pode ter o valor do código ASCII do carater **R** ou do carater **N**, i.e., $\{82\}_1$ ou $\{78\}_1$
$<S_1\|S_2\|…\|S_N>*$	Uma sequência de símbolos, de entre um grupo de símbolos possíveis, guardados como uma sequência dos códigos ASCII respetivos; ; nenhum dos símbolos pode ser um dos símbolos especiais **<@>** ou **<;>**; por exemplo, **<0|1>*** indica que no ficheiro pode aparecer uma string representando uma sequência binária, ou seja, uma sequência de carateres **0** ou **1**, i.e., uma sequência de bytes com o valor $\{48\}_1$ ou $\{49\}_1$
@	Símbolo especial que serve de separador de campos num ficheiro de saída e que deve ser guardado como o código ASCII equivalente, i.e., $\{64\}_1$
;	Símbolo especial que serve de separador de campos num ficheiro de saída e que deve ser guardado como o código ASCII equivalente, i.e., $\{59\}_1$

Processamento por blocos

Numa ferramenta que precise processar os dados dum ficheiro completo é natural que os dados sejam lidos do ficheiro para processamento em memória uma vez que o processamento em ficheiro é mais complexo e muito menos eficiente.

Assim, os módulos desta ferramenta devem ler os dados do ficheiro de entrada em blocos para um *buffer* em memória onde depois irão ser processados, um bloco de cada vez. Os resultados do processamento de cada bloco também devem ser guardados num outro *buffer* em memória que só deve ser gravado num ficheiro de saída quando o bloco completo tiver sido processado. Ou seja, a utilização de *buffers* em memória deve ser feita tanto para armazenamento temporário dos dados do ficheiro de entrada a processar como para armazenamento temporário dos resultados desse processamento.

O tamanho de cada bloco define-se, por defeito, como 64Kbytes. Outros três tamanhos podem ser usados por indicação expressa do utilizador: 640Kbytes, 8Mbytes ou 64Mbytes. O último bloco a processar poderá ter um tamanho inferior aos outros blocos, dependendo do número de bytes que restam para processar no ficheiro. Opcionalmente, a ferramenta, por questões de eficiência, poderá não permitir que este último bloco tenha menos do que 1Kbyte. Neste caso, o penúltimo bloco deve incluir também estes dados restantes do ficheiro.

Assim pode definir-se um algoritmo genérico (Figura 2) que expresse em abstrato a estratégia de processamento por blocos que os módulos da ferramenta devem fazer:

```
f_e=ficheiro_entrada()
f_s=ficheiro_saida()
tamanho_bloco_e=definir_tamanho_bloco_entrada()
tamanho_ficheiro=calcular_tamanho_ficheiro(f_entrada)
numero_blocos=tamanho_ficheiro/tamanho_bloco_e
fazer
  { buffer_e=alocar_espaço_memoria(tamanho_bloco_e)
    n_bytes_e=ler_dados_ficheiro(f_e,tamanho_bloco_e,buffer_e)
    tamanho_bloco_s=definir_tamanho_bloco_saida(n_bytes)
    buffer_s=alocar_espaço_memoria(tamanho_bloco_s)
    n_bytes_s=processar_bloco(buffer_e,n_bytes_e,buffer_s)
    escrever_dados_ficheiro(f_s,n_bytes_s,buffer_s)
    libertar_espaço_memoria(buffer_e)
    libertar_espaço_memoria(buffer_s)
    numero_blocos=numero_blocos-1
  }
```

Figura 2: Algoritmo genérico de estratégia de processamento por blocos.

A Figura 3 apresenta um exemplo esquemático da divisão em blocos de 64Kbytes dum ficheiro de entrada de 216Kbytes realizada no módulo da primeira fase:

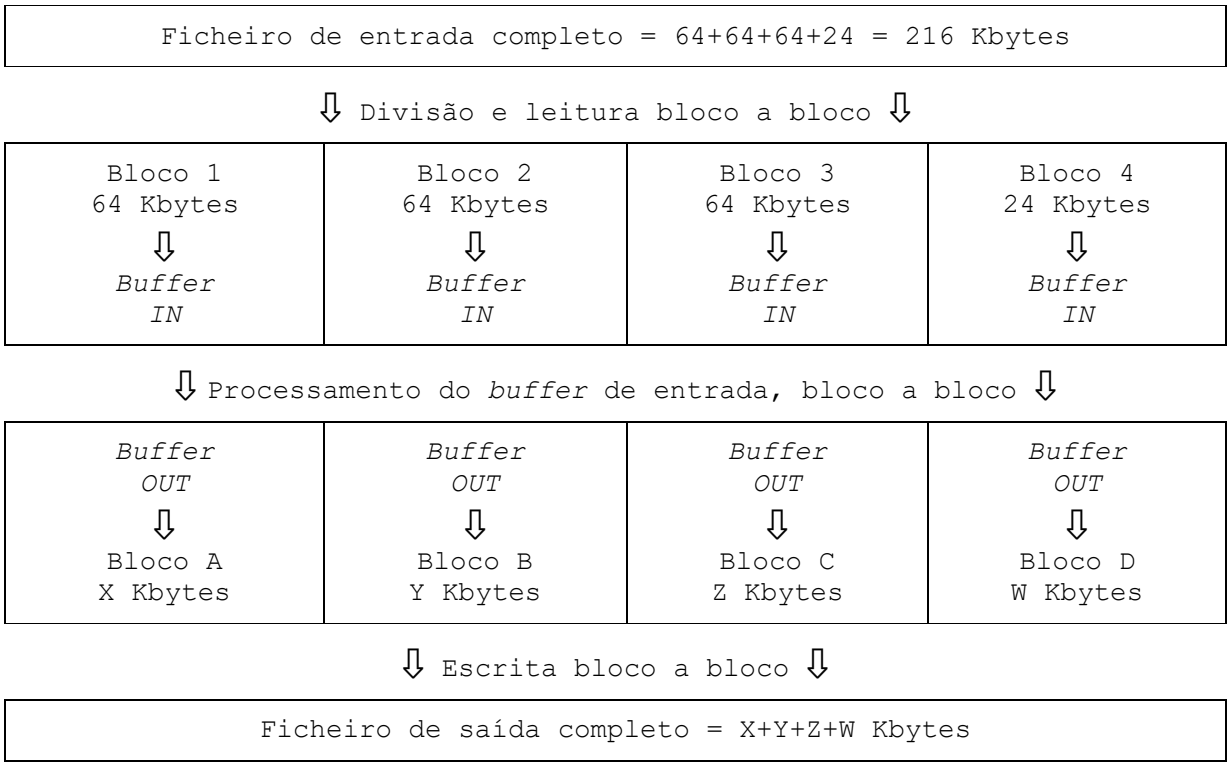

Figura 3: Exemplo de processamento por blocos num ficheiro de 216Kbytes.

De notar que o tamanho do *buffer* de saída (que vai conter os resultados do processamento dos dados no *buffer* de entrada) não será igual ao tamanho do buffer de entrada. O seu tamanho depende da função específica implementada no processamento (dependente do módulo).

Adicionalmente, apenas o módulo da primeira fase é que permite ao utilizador definir o tamanho dos blocos (ou aceitar o tamanho por defeito) para processar o ficheiro de entrada original. Todos os outros módulos devem utilizar os tamanhos dos blocos já definidos nos próprios ficheiros de entrada pois estes são já o resultado da execução de módulos anteriores. Ou seja, depois da primeira fase, os módulos utilizam como tamanho dos blocos o tamanho dos blocos que resultaram do processamento do ficheiro de entrada do módulo anterior. Esse tamanho, que vem indicado no próprio ficheiro, pode até ser diferente para todos os blocos.

De salientar que uma outra vantagem da estratégia de processamento por blocos é a maior facilidade de implementar otimização por processamento paralelo dos blocos através da utilização de *multi-threading* (ver secção de "Otimização por execução paralela").

Em resumo, podem fazer-se as seguintes considerações sobre a divisão por blocos dos dados de entrada e saída:

- Cada módulo do trabalho é como um programa separado que processa uma quantidade de informação que está num ou mais ficheiros, designados por ficheiros de entrada.

- Cada módulo tem de processar a informação dos seus ficheiros de entrada e produzir um resultado que vai para um ou mais ficheiros de saída.

- Existem dois grupos de ficheiros, os ficheiros com dados do utilizador (no formato original ou processados com algoritmos de compressão ou descompressão) e ficheiros com meta-dados e que são gerados pelos módulos para gravar informação adicional que os módulos precisam trocar entre si.

- Os ficheiros de dados são os ficheiros originais que o utilizador quer comprimir ou os ficheiros do tipo **rle** (ficheiros que contêm a mesma informação dos ficheiros originais, mas comprimida com o algoritmo RLE) ou os ficheiros do tipo **shaf** (ficheiros que contêm a mesma informação dos ficheiros originais ou dos ficheiros **rle** mas comprimida com o algoritmo Shannon-Fano).

- Os ficheiros de meta-dados são os ficheiros do tipo **freq** ou **cod** e contêm informação adicional que é necessária para passar meta-informação entre os módulos. Por exemplo, os ficheiros tipo **freq**, gerados pelo módulo/programa da fase A, têm informação sobre a frequência dos símbolos nos ficheiros originais ou nos ficheiros do tipo **rle**, que é necessária para depois o módulo/programa da fase B poder gerar os códigos Shannon-Fano.

- O processamento da informação dos dados ou dos meta-dados não deve ser feita nos próprios ficheiros pois seria muito pouco eficiente, problemática para ficheiros grandes, impossível para ficheiros enormes. Os ficheiros são apenas repositórios de informação. Os programas informáticos que fazem compressão/descompressão de ficheiros não podem tratar/processar os dados diretamente nos ficheiros. Primeiro têm que ler os dados dos ficheiros de entrada para memória, depois processar os dados que estão em memória, guardar os resultados do processamento também em memória e, por fim, guardar os resultados que estão em memória para ficheiros de saída.

- As funções de leitura de dados dos ficheiros são muito lentas, comparativamente com as funções que manipulam dados em memória. Portanto, os programas devem executar o menor número possível de funções que leiam ou gravem dados dos ficheiros. Nesse sentido, sempre que se lerem dados dos ficheiros de entrada deve ler-se uma quantidade grande de dados duma só vez para minimizar o número de vezes que a

função de leitura do ficheiro é executada. Um bloco é essa quantidade de dados (normalmente identificados como bytes ou símbolos neste tipo de programas de compressão) que são lidos duma só vez do ficheiro (numa única chamada à função de leitura do ficheiro) e colocados num *buffer* em memória.

- Os dados de entrada dum bloco são armazenados temporariamente num *buffer* de entrada e são processados em memória. Do seu processamento resultam outros dados. Cada módulo faz um processamento diferente dos dados de entrada. Cada módulo gera resultados diferentes.

- Os dados que resultam do processamento dos dados do *buffer* de entrada são gravados num *buffer* temporário de saída. Quando o processamento de todos os dados do *buffer* de entrada estiver terminado o *buffer* de saída tem os resultados completos. Os dados do *buffer* de saída formam agora um novo bloco de dados com um tamanho que depende diretamente do tamanho definido para o bloco de dados de entrada. Esse bloco de dados de saída deve ser, finalmente, guardado num ficheiro de saída, numa única operação de escrita.

- Idealmente, deveria ler-se todos os dados dum ficheiro de entrada duma só vez para um *buffer* de entrada. No entanto, a memória disponível para os programas é limitada e essa estratégia seria impossível ou muito pouco eficiente para ficheiros grandes ou em dispositivos com limitações de *hardware* ou *software*. A solução é dividir os ficheiros em blocos de dados suficientemente grandes para minimizar as operações de leitura dos ficheiros. Essa divisão é apenas em termos de quantidade de bytes que cabem num bloco, não é uma divisão que tenha em consideração a semântica ou organização dos dados dos ficheiros. Define apenas o número de bytes que são lidos duma só vez para memória. Por exemplo, se um ficheiro tem T bytes de tamanho, se se definir o tamanho de cada bloco como K bytes, o número de blocos (valor inteiro) que têm de ser lidos do ficheiro é igual a T/K+1 (ou T/K no caso de T ser um múltiplo de K). Quando T não é múltiplo de K, o último bloco tem o tamanho igual a T-T/K.

- No material fornecido de apoio ao trabalho prático inclui-se o código duma função em C para calcular o número de blocos e o tamanho do último bloco tendo em consideração um determinado ficheiro e um determinado tamanho de bloco. Esta função é escrita em C *standard* e é independente do sistema operativo.

- Neste trabalho define-se, por defeito, um tamanho de bloco para ficheiros originais (ficheiros do utilizador) de dados de entrada como sendo 64Kbytes (opcionalmente é possível que o utilizador defina, no módulo da fase A ou no módulo da fase D, este tamanho como sendo antes de 640Kbytes, 8Mbytes ou 64Mbytes). Dependendo dos recursos de memória

disponíveis, quanto maior for o tamanho dos blocos mais rápido e eficiente vai ser o programa.

- O tamanho dos blocos só pode ser definido no módulo da fase A ou, em condições especiais, no módulo da fase D. Nos módulos das outras fases o tamanho dos blocos depende precisamente da definição do tamanho dos blocos feita na fase A, sendo que essa informação é passada para os módulos das outras fases através dos ficheiros de meta-dados (do tipo **freq** ou do tipo **cod**) ou de ficheiros de dados (do tipo **shaf**).

- Note-se que um tamanho definido no módulo da fase A pode não se manter igual de fase para fase porque os dados são processados e o tamanho dos resultados do processamento podem ter um tamanho diferente dos dados do bloco de entrada. Por exemplo, um ficheiro de dados do tipo **rle** ou do tipo **shaf** terá blocos de tamanho mais pequeno do que o tamanho definido inicialmente da fase A para dividir o ficheiro de dados original. A compressão dos dados nos blocos de entrada faz com que os blocos de saída tenham, normalmente, um tamanho inferior.

- As considerações feitas sobre a minimização do número de funções de leitura de dados dum ficheiro de entrada também se aplicam à minimização do número de funções de escrita de dados dum ficheiro de saída.

A. Módulo para cálculo das frequências dos símbolos e compressão RLE

No primeiro passo da primeira fase o programa deve ler do ficheiro de entrada um bloco de tamanho máximo de 64Kbytes (valor por defeito), 640Kbytes, 8Mbytes ou 64Mbytes, e verificar se compensa fazer uma pré-compressão simples utilizando o mecanismo *Run-Lenght Encoding* (RLE). Para isso deve ler-se o bloco para memória e comprimi-lo utilizando o mecanismo RLE. Se o bloco obtido tiver uma compressão superior a 5% então deve utilizar-se a compressão RLE também para o resto dos blocos do ficheiro original, i.e., deve processar-se o ficheiro original por blocos, um de cada vez, comprimindo os dados dos blocos e guardando o resultado, bloco a bloco, num ficheiro de saída do tipo `rle`. Por uma questão de desempenho, recomenda-se que se faça, para cada bloco e em simultâneo, a contagem dos símbolos e a compressão RLE.

De notar que o tamanho do último bloco pode ser inferior ao tamanho definido para os outros blocos, uma vez que, em geral, o tamanho total do ficheiro não é um múltiplo do tamanho escolhido para os blocos. Além disso, se o tamanho dum último for inferior a 1Kbyte, então este bloco deve ser absorvido pelo penúltimo bloco, passando os dois últimos blocos a um único último bloco com um tamanho um pouco maior do que o tamanho definido para os blocos. Por fim, deve definir-se como 1Kbyte o limite mínimo para o tamanho total dum ficheiro para que a ferramenta possa ser executada. Se o ficheiro de entrada for inferior a 1Kbyte a ferramenta deve cancelar a execução e avisar o utilizador.

Compressão RLE

A compressão RLE ajuda em ficheiros em que a distribuição dos símbolos no ficheiro original é por molhos, i.e., em que existem grandes agrupamentos de símbolos iguais consecutivos. O algoritmo RLE pressupõe o processamento dum bloco de símbolos do ficheiro original e por cada sequência do mesmo símbolo deve gerar-se o padrão RLE $\{0\}_1\{\text{símbolo}\}_1\{\text{número\_de\_repetições}\}_1$, em que o número de repetições é um valor entre 1 e 255. Este padrão indica que no ficheiro original apareceriam entre 1 e 255 símbolos iguais seguidos. Como o comando RLE ocupa 3 bytes, só compensa usa-lo quando um símbolo se repete pelo menos 4 vezes.

A exceção é o próprio símbolo **{0}**. Para este símbolo, independentemente do número de vezes que apareça, deve ser sempre gerado o padrão RLE e o número de repetições é um valor entre 1 e 255, indicando que no ficheiro original apareceriam entre 1 e 255 símbolos **{0}** seguidos.

Se o primeiro bloco dum ficheiro tiver a seguinte sequência de 76 símbolos:
aa936c$\{0\}_1$g648bbbbbbbbbbbbbbbbakjdaaajdf99999999999999999999999999999999fsissyi[…]
então a compressão RLE originaria a seguinte sequência de 35 símbolos com 4 padrões RLE (conseguindo-se uma compressão de $\frac{76-36}{76} = 53\%$):
aa936c$\{0\}_1\{0\}_1\{1\}_1$g648$\{0\}_1$b$\{17\}_1$akjdaaajdf$\{0\}_1$9$\{31\}_1$fsissyi[…]
Esta sequência comprimida deve então ser analisada para se contarem o número de vezes que aparecem os símbolos (já com os padrões RLE incluídos).

Se a compressão RLE obtida no primeiro bloco for inferior a 5% então a compressão RLE deve ser ignorada, nenhum ficheiro do tipo `rle` deve ser

gerado e deve passar-se imediatamente ao passo seguinte da primeira fase.

De notar que, se o utilizador desejar, pode forçar a utilização do mecanismo de compressão RLE através dum argumento opcional, mesmo que a compressão do primeiro bloco seja inferior a 5%.

Cálculo das frequências

No segundo passo desta primeira fase devem calcular-se as frequências dos símbolos no ficheiro de entrada. Se no primeiro passo tiver sido utilizada a compressão RLE, resultando daí um ficheiro do tipo **rle**, então o cálculo das frequências dos símbolos deve ser feito sobre o ficheiro **rle**. Opcionalmente (fica a cargo dos alunos decidirem a estratégia que preferem), o módulo pode gerar sempre um ficheiro do tipo **freq** com o resultado da análise de ocorrências para o ficheiro original, mesmo nos casos em que também gere um ficheiro do tipo **freq** para análise de ocorrências para o ficheiro **rle**.

Cada bloco do ficheiro de entrada (o original ou o **rle**) deve ser lido e percorrido byte a byte por forma a contar quantas vezes aparece cada símbolo (ou valor do byte). No final da análise de cada bloco deve ser gerada uma lista implícita das frequências e escrita no ficheiro de saída do tipo **freq**.

O ficheiro de saída do tipo **freq** terá a seguinte sintaxe:

@<R|N>@[número_de_blocos]@[tamanho_bloco_1]@[frequência_símbolo_0_bloco_1];[frequência_símbolo_1_bloco_1];[…];[frequência_símbolo_255_bloco_1]@[tam_bloco_2]@[frequência_símbolo_0_bloco_2];[frequência_símbolo_1_bloco_2];[…];[frequência_símbolo_255_bloco_2]@[…]@0

Ou seja, no início deve ser sinalizado se a análise das frequências foi realizada sobre o ficheiro original (quando não foi aplicada compressão RLE deve incluir-se o símbolo **N**) ou sobre o ficheiro resultante da compressão RLE do ficheiro original (deve incluir-se o símbolo **R**). Em seguida vem a lista de frequências dos símbolos para cada bloco. Em cada bloco, deve ser guardado o tamanho do bloco em bytes e depois uma lista de 255 valores que representam a frequência dos 255 símbolos possíveis; esta lista começa com o número de vezes que ocorre o byte com valor 0, seguido do número de vezes que ocorre o byte com o valor 1 e assim por diante até ao número de vezes que ocorre o byte com o valor 255. Um valor de tamanho de bloco igual a zero indica que já não há mais blocos. O último bloco tem sempre os bytes restantes até final do ficheiro enquanto os outros blocos têm sempre o mesmo tamanho. A inexistência de valor de frequência quer dizer o mesmo valor de frequência do símbolo anterior.

Segue-se um exemplo da sintaxe dum ficheiro **freq** resultante desta fase:
@R@2@57444@1322;;335[…];456@1620@19;21;[…];6@0

Neste exemplo, o ficheiro resultante da compressão RLE tem 59064 bytes e foi analisado em dois blocos, um com 57444 bytes e outro com 1620. No primeiro bloco aparece 1322 vezes o byte com o valor 0, também 1322 vezes o byte com o valor 1 (a não existência de valor representa implicitamente o mesmo valor de frequência do símbolo anterior), 335 vezes o byte com o valor 2,…, e 456 vezes o byte com o valor 255. No segundo e último bloco

aparece 19 vezes o byte com o valor 0, 21 vezes o byte com o valor 1,…, e 6 vezes o byte com o valor 255. De notar que os blocos originais que foram processados no algoritmo RLE tinham, respetivamente, 65536 bytes (64 Kbytes) e 2013 bytes, ou seja, o tamanho do ficheiro original era de 67549 bytes.

Opções e ficheiros deste módulo

Pode resumir-se o conjunto de opções, ficheiros de entrada e saída e texto de saída na consola na seguinte lista:

- Argumento do programa para executar este módulo: `-m f`
- Opção para indicar tamanho dos blocos para análise (640Kbytes, 8Mbytes ou 64Mbytes, respetivamente; por defeito, o tamanho é 64Kbytes):
  `[-b K|m|M]`
- Opção para forçar compressão RLE: `[-c r]`
- Ficheiro de entrada: de qualquer tipo e serve de fonte de informação, sendo que cada símbolo é um bloco de oito bits
- Texto de saída na consola:
  [identificação do(s) autor(es), curso, data, etc.]
  [identificação do módulo]
  [número de blocos]
  [tamanhos dos blocos processados no ficheiro original, em bytes]
  [ficheiro resultante do mecanismo RLE e respetiva taxa de compressão]
  [tamanhos de todos os blocos processados no ficheiro RLE, em bytes]
  [tempo de execução do módulo em milissegundos]
  [ficheiros gerados]
  Os tamanhos dos blocos do ficheiro original a indicar são o tamanho de todos os blocos normais e o tamanho do último bloco.
  A eventual indicação do nome do ficheiro resultante da compressão RLE deve ser acompanhado do valor da taxa de compressão global para todo o ficheiro.
  Os tamanhos dos blocos do ficheiro original a indicar são o tamanho de todos os blocos normais e o tamanho do último bloco.
  De notar que a contagem do tempo de execução deve terminar antes de o programa fazer a escrita na consola destas mensagens.
- Ficheiros de saída: um ficheiro que tenha a lista das frequências de cada símbolo em cada bloco. Quando for gerado um ficheiro `rle` deve ser gerado um ficheiro `freq` resultante da análise de ocorrências dos símbolos no ficheiro `rle` e, opcionalmente, pode também ser gerado um ficheiro `freq` com a análise do de ocorrências dos símbolos no ficheiro original (neste caso são gerados dois ficheiros `freq`).
- Terminação dos ficheiros de saída:
  `.rle` (caso tenha sido gerado um ficheiro com compressão RLE)
  `.freq` (ficheiro com as frequências dos símbolos no ficheiro original e/ou, se existir, no ficheiro `.rle`)

Exemplo dum comando para executar este módulo sobre o ficheiro de entrada **exemplo.txt** e utilizando blocos de 64Kbytes para o ficheiro original:

```
Shell> shafa exemplo.txt -m f -c r
```

A sua execução deve gerar um ficheiro de saída **exemplo.txt.rle** e dois ficheiros **freq**, **exemplo.txt.freq** e **exemplo.txt.rle.freq**, (ainda que só seja obrigatório a criação do segundo). O seguinte texto seria enviado para a consola:

```
John Doe, a12234, MIEI/CD, 1-jan-2021
Módulo: f (cálculo das frequências dos símbolos)
Número de blocos: 2
Tamanho dos blocos analisados no ficheiro original: 65536/2013 bytes
Compressão RLE: exemplo.txt.rle (13% compressão)
Tamanho dos blocos analisados no ficheiro RLE: 57444/1620 bytes
Tempo de execução do módulo (milissegundos): 12
Ficheiros gerados: exemplo.txt.freq, exemplo.txt.rle.freq
```

B. Módulo para cálculo das tabelas de codificação

O módulo para implementar esta segunda fase deve ter como único ficheiro de entrada um ficheiro do tipo `freq` e deve ter como saída um ficheiro do tipo `cod` que conterá a tabela de códigos Shannon-Fano calculados a partir dos valores de frequência dos símbolos no ficheiro de símbolos (do ficheiro original ou do ficheiro `rle`). No final da execução o módulo deve imprimir na consola o texto de saída com informações sobre a dita execução.

Construir a tabela de codificação

O principal objetivo deste módulo é a construção duma tabela de códigos para cada bloco resultante da fase anterior e que está especificado no ficheiro `freq`. Embora o algoritmo de SF refira a utilização da probabilidade dos símbolos aparecerem na fonte, o uso alternativo da frequência dos símbolos calculada num determinado intervalo temporal ou num espaço local de armazenamento (ficheiros) produz o mesmo efeito.

Assim, o primeiro passo será ordenar a lista de símbolos na tabela pelo valor descendente da sua frequência. Depois o algoritmo SF deve seguir normalmente, apenas considerando o valor das frequências em vez de valores de probabilidades.

A definição do algoritmo de SF adapta-se bem a utilização de mecanismos de recursividade. Embora a recursividade seja uma elegante formulação ao nível matemático e algorítmico, a sua implementação pode trazer problemas de escalabilidade na sua implementação computacional por causa do crescimento da *stack* de chamadas consecutivas em *loop* à mesma função. No caso da implementação do algoritmo de SF, a recursividade pode ser usada desde que a programação seja cuidada e que se garante um número limitado de chamadas recursivas consecutivas, o que depende em larga escala da cardinalidade da fonte, i.e., do número de símbolos do alfabeto. No caso particular deste programa, a cardinalidade da fonte é igual a 256 pelo que o problema da recursividade pode não ser importante. É recomendado, de qualquer forma, que a recursividade pura no algoritmo seja substituída, na implementação, por mecanismos não recursivos que atingem o mesmo objetivo.

A Figura 4 apresenta um excerto do algoritmo de codificação SF usando recursividade, incluindo uma função para cálculo da divisão de cada grupo de símbolos para minimizar a soma das frequências:

```
calcular_melhor_divisão(freq[],i,j)
  { div=i, g1=0
    total=mindif=dif=soma(freq[],i,j)
    repetir
      { g1=g1+freq[div]
        dif=abs(2*g1-total)
        se (dif<mindif) então
          { div=div+1
            mindif=dif
          }
        senão dif=mindif+1
      } até (dif!=mindif)
    calcular_melhor_divisão=div-1
  }
```

```
calcular_codigos_SF(freqs[],codes[],start,end)
  { se (start!=end) então
      { div=calcular_melhor_divisão(freqs[],start,end)
        add_bit_to_code('0',codes[],start,div)
        add_bit_to_code('1',codes[],div+1,end)
        calcular_codigos_SF(freqs,start,div)
        calcular_codigos_SF(freqs,div+1,end)
      }
  }
```

Figura 4: Algoritmo para cálculo de códigos SF.

Representação dos códigos no ficheiro cod

Depois da execução do algoritmo de SF a cada bloco é necessário armazenar o resultado (os códigos binários para todos os símbolos) num ficheiro do tipo **cod**. Este ficheiro conterá a respetiva tabela de códigos para cada bloco de acordo com a seguinte sintaxe:

@<R|N>@[número_de_blocos]@[tamanho_bloco_1]@<0|1>*;[…];<0|1>*@[tamanho_bloco_2]@<0|1>*;[…];<0|1>*@[…]@0

Ou seja, no início deve ser sinalizado se a análise das frequências foi realizada sobre o ficheiro original (copia-se o valor deste campo do ficheiro **freq**) e o número de blocos. Em seguida, para cada bloco teremos guardado o tamanho do bloco em bytes (mesmo valor que está guardado nos ficheiros **freq**) seguido da lista completa dos códigos de todos os 255 símbolos do alfabeto possível (primeiro o código do símbolo (ou byte) com o valor 0, depois o código do símbolo com o valor 1 e assim por diante até ao código do símbolo com o valor 255. Os códigos são *strings* binárias. Um valor de tamanho de bloco igual zero indica que já não há mais blocos. De notar que se um determinado símbolo tem uma frequência igual a zero então não vai ter código SF correspondente e, nesse caso, a sequência binária correspondente é nula/vazia e aparecerão dois símbolos separadores consecutivos. Por exemplo, no seguinte excerto @R@2@57444@10100;;;00111;[…], o código SF do símbolo com o valor 0 é **10100**, os códigos dos símbolos com os valores 1 e 2 são vazios/nulos (porque a frequência dos símbolos era igual a zero) e o código do símbolo com o valor 3 é **00111**.

Salienta-se também que a lista de códigos é implicitamente ordenada por valor do símbolo, tal como para os ficheiros de frequências. Ou seja, o primeiro código é relativo ao símbolo com o valor 0, o segundo código é relativo ao símbolo com o valor 1 e assim por diante. A lista destes códigos num ficheiro **cod** não está ordenada por tamanho de código ou probabilidade/frequência dos símbolos, tal como acontece nos ficheiros **freq**.

Segue-se um exemplo da sintaxe dum ficheiro **cod** resultante desta fase:
@R@2@57444@0001001;00111101;10110111;[…];1000110111@1620@010111;110111101111011;[…];0110111000101@[…]@0

Neste exemplo, o ficheiro resultante da compressão RLE tem 67549 bytes e foi anteriormente analisado em dois blocos, um com 64Kbytes (65536 bytes)

e outro com 2013 bytes. Do cálculo da tabela de SF para o primeiro bloco resultou o código binário **0001001** para o símbolo (byte) com o valor 0 e **00111101** para o o símbolo com o valor 1, o código binário **10110111** para o símbolo com o valor 2,…, e o código binário **1000110111** para o símbolo com o valor 255. Do cálculo da tabela de SF para o segundo e último bloco com 2013 bytes resultou o código binário **010111** para o símbolo (byte) com o valor 0, o código binário **110111101111011** para o símbolo com o valor 1,…, e o código binário **0110111000101** para o símbolo com o valor 255.

De notar que a inclusão dos campos referentes à indicação da utilização eventual da compressão RLE e do tamanho dos blocos é copiada dos ficheiros **freq**. Isto porque o módulo da fase seguinte de codificação não necessita analisar os ficheiros tipo **freq**, apenas o ficheiro **cod** e o ficheiro original (ou ficheiro **rle** se foi utilizada compressão RLE).

Opções e ficheiros deste módulo

Pode resumir-se o conjunto de opções, ficheiros de entrada e saída e texto de saída na consola na seguinte lista:

- Argumento do programa para executar este módulo: **-m t**
- Ficheiro de entrada: ficheiro do tipo **freq**
- Texto de saída na consola:
  [identificação do(s) autor(es), curso, data, etc.]
  [identificação do módulo]
  [número de blocos]
  [tamanhos dos blocos processados no ficheiro de símbolos, em bytes]
  [tempo de execução do módulo em milissegundos]
  [ficheiro gerado]
  O ficheiro de símbolos pode ser o ficheiro original ou o ficheiro **rle** caso a compressão RLE tenha sido utilizada.
  De notar que a contagem do tempo de execução deve terminar antes de o programa fazer a escrita na consola destas mensagens.
- Ficheiro de saída: um ficheiro que tenha a lista dos códigos dos símbolos para cada bloco.
- Terminação dos ficheiros de saída:
  cod (ficheiro com a lista dos códigos dos símbolos para cada bloco)

Exemplo dum comando para executar este módulo sobre o ficheiro de entrada **exemplo.txt** e utilizando blocos de 64Mbytes:

```
Shell> shafa exemplo.txt.rle.freq -m t
```

A sua execução deve gerar um ficheiro de saída **exemplo.txt.rle.cod** e, por exemplo, o seguinte texto na consola:

```
John Doe, a12234, MIEI/CD, 1-jan-2021
Módulo: t (cálculo dos códigos dos símbolos)
Número de blocos: 2
Tamanho dos blocos analisados no ficheiro de símbolos: 57444/1620 bytes
Tempo de execução do módulo (milissegundos): 296
Ficheiro gerado: exemplo.txt.rle.cod
```

C. Módulo para codificação do ficheiro original/RLE

O módulo para implementar as funcionalidades desta terceira fase deve ter como ficheiros de entrada um ficheiro do tipo **cod** e o ficheiro onde está a sequência de símbolos a codificar/comprimir: ou o ficheiro original ou o ficheiro **rle** correspondente se foi utilizado previamente o mecanismo RLE. O módulo deve gerar como saída um ficheiro do tipo **shaf** que conterá apenas a sequência binária de códigos SF calculados na fase anterior. Mais uma vez, no final da execução o módulo deve imprimir na consola o texto de saída com informações sobre a dita execução.

Construir a sequência codificada de bits

O principal objetivo deste módulo é, para cada bloco, a geração duma sequência de bits que seja o resultado da concatenação dos códigos binários dos símbolos que os códigos vão substituir. Esta operação é a mais determinante para o desempenho global das ferramentas de codificação baseadas em mecanismos estatísticos. A operação repetitiva de serialização dos códigos binários numa sequência única deve ser implementada utilizando operações de execução rápida num computador e devem ser evitadas estratégias que utilizem operações matemáticas complexas e/ou de implementação mais lenta ou operações de manipulação de valores em memória que sejam pouco eficientes ou lentas.

A operação de codificação ou compressão deve ser executada para cada bloco e o resultado deve ser armazenado temporariamente em memória. Tal como na compressão RLE, apenas no final da codificação do bloco é que o resultado é gravado em ficheiro de saída, neste caso no ficheiro **shaf** correspondente. O resultado da codificação dum bloco será uma sequência de bits que pode não ser múltipla de oito, ou seja, a sequência binária resultante pode não acabar na fronteira dum byte. No entanto, só é possível gravar para ficheiro quantidades de bits múltiplas de oito (i.e., bytes inteiros) pelo que é possível que, na maior parte dos casos, a sequência de bits resultante da codificação acabe algures antes do final do último byte. Nestes casos, o valor dos bits que ficam para lá do final do último código concatenado até final do último byte é irrelevante (*padding*). Quando for executada a operação inversa de descodificação/descompressão do bloco, esses bits serão ignorados.

No princípio do ficheiro **shaf** deve ser acrescentado um cabeçalho com a indicação do número blocos e do tamanho em bytes de cada bloco resultante da codificação SF para todos os blocos. Esta informação serve para otimizar a leitura dos blocos para processamento.

A sintaxe do ficheiro de saída do tipo **shaf** é a seguinte:
`@[número_de_blocos]@[tamanho_bloco_1]@(sequência_de_bits_resultante_da_codificação_SF_do_bloco_1)@[…]@[tamanho_último_bloco]@(sequência_de_bits_resultante_da_codificação_SF_do_último_bloco)`

Segue-se um extrato do ficheiro de saída **shaf** se o ficheiro de símbolos de entrada começar com a seguinte sequência de símbolos `aabbcc[…]` e os códigos binários SF para os símbolos **a**, **b** e **c** são, respetivamente, **00**, **010** e **011**. Parta-se do princípio que o ficheiro foi processado em dois blocos que,

após, codificação SF ficaram com os tamanhos 3962 e 1210, respetivamente: @2@3962@(00)(00)(010)(010)(011)(011)[…]@1210@[…]

Otimização da codificação binária SF com matrizes de bytes

A codificação da sequência de símbolos de entrada pode ser otimizada utilizando funções de manipulação de *arrays* de bytes sem haver necessidade de implementar algoritmos que utilizam funções matemáticas mais lentas e que tratam/processam bits individuais.

Apresenta-se em seguida uma estratégia de otimização do desempenho desta operação tão importante deste tipo de ferramentas. Esta proposta serve apenas de recomendação, cabendo aos alunos a decisão da implementação desta ou doutra estratégia, otimizada ou não. Em qualquer dos casos, os alunos só devem implementar algoritmos que percebam inteiramente.

Parta-se do princípio que a sequência original a codificar já está armazenada num array de bytes e que a tabela de códigos também já está calculada e os dados associados também já estão armazenados em arrays, conforme segue no exemplo da Figura 5.

N=4 símbolos, CODE_MAX_SIZE=2 bytes

	symbol	freq.	codes	index (bytes)	next (bits)	
offset=0	**0**	100	**0**0000000.00000000	0	1*N	N x 8 = 32
	1	25	**110**00000.00000000	0	3*N	
	2	25	**111**00000.00000000	0	3*N	
	3	50	**10**000000.00000000	0	2*N	
offset=1	**0**	100	0**0**000000.00000000	0	2*N	
	1	25	0**110**0000.00000000	0	4*N	
	2	25	0**111**0000.00000000	0	4*N	
	3	50	0**10**00000.00000000	0	3*N	
	[…]					
offset=7	**0**	100	0000000**0**.00000000	1	0	
	1	25	0000000**1**.**10**000000	1	2*N	
	2	25	0000000**1**.**11**000000	1	2*N	
	3	50	0000000**1**.**0**0000000	1	1*N	

Figura 5: Dados na estratégia de otimização por matriz.

A informação das últimas três colunas da tabela (**codes**, **index**, **next**) podem ser diretamente implementadas por um *array* bidimensional de bytes. A primeira coluna com os símbolos não é de implementação obrigatória porque os símbolos estão implícitos na posição do *array*. A informação do símbolo zero aparece na posição zero do array e assim por diante.

No exemplo apresentado são utilizados apenas quatro símbolos (0,1,2,3) por uma questão facilidade de representação. Na implementação da ferramenta **shafa**, a cardinalidade é **N**=256, para símbolos de oito bits (um byte).

Depois de se obter os códigos normais retirados do ficheiro `cod`, tem que se formatar a sequência de bits do códigos especiais em **codes** acrescentando bits a zero ao código mais comprido até se ficar com um número de bits total que seja múltiplo de oito. Acrescentam-se depois ainda mais oito bits. O número de bits com que o maior código assim fica é **CODE_MAX_SIZE** (o tamanho desse código é sempre um múltiplo de oito bits por isso vem em bytes) e é o número de bits com que todos os outros códigos têm de ficar, acrescentando-se a todos os códigos os bits necessários até ficarem com o tamanho de **CODE_MAX_SIZE** bytes. Todos os bits de sufixo ficam a zero. Esta coluna dos códigos pode ser diretamente implementada como parte do *array* bidimensional de bytes já referido.

O segundo passo é inserir os valores das duas últimas colunas. O valor da coluna **index** representa o índice do byte, na sequência total de bytes do código já com prefixos e sufixos do passo anterior, onde deverá estar o bit real do código do próximo símbolo a codificar, i.e., o primeiro bit que não faz parte do sufixo.

O valor da coluna **next** representa o índice do primeiro bit do sufixo no byte respetivo, multiplicado por **N**. Se um determinado código não tiver bits de sufixo, o valor de **next** é zero.

Relembra-se que a tabela deve estar ordenada por valor do símbolo, conforme se mostra no exemplo, e que já é a forma como os códigos estão ordenados nos ficheiros `cod`.

Estes três passos anteriores servem para criar e formatar a primeira tabela para o **offset**=0 em que os códigos reais começam no primeiro bit das sequências de codificação da coluna **codes**.

Em seguida copia-se o conteúdo desta tabela (primeiras **N** linhas da tabela final) para o grupo seguinte de linhas **N**+1 até 2*__N__ mas fazendo um *shift* à direita (acrescentado um bit de prefixo e retirando um bit do sufixo) ao conteúdo da coluna **codes**.

Atualizam-se as colunas **index** e **next** das linhas **N**+1 a 2*__N__ em acordo. Esta sub-tabela será a correspondente ao **offset**=1 (o valor de **offset** representa o número de *shifts* executados para criar o conteúdo da coluna **codes** a partir da sub-tabela inicial com **offset**=0).

Repete-se o processo anterior até **offset**=7. No final obtém-se uma tabela completa com **N***8 linhas.

Na Figura 6 fica um algoritmo da função de codificação, tendo em atenção que a função OR(byte_a,byte_b) é uma função booleana de aplicação binária (aplicada bit a bit aos dois argumentos):

```
binary_coding(symbols[],number_of_symbols,codes[],index[],next[])
  { offset=0, ind_in=0, ind_out=0
    while ind_in<number_of_symbols
           { symbol=symbols[ind_in]+offset
        n_bytes_in_code=index[symbol]
        code=codes[symbol]
        ind_code=0
        while ind_code≤n_bytes_in_code
          { coded_sequence[ind_out]=OR(coded_sequence[ind_out],code[ind_code])
            if ind_code<n_bytes_in_code then
              { ind_out=ind_out+1
                ind_code=ind_code+1
              }
          }
             offset=next[symbol]
             ind_in=ind_in+1
           }
    binary_coding=coded_sequence
  }
```

Figura 6: Algoritmo da função de codificação binária por matriz.

Opções e ficheiros deste módulo

Pode resumir-se o conjunto de opções, ficheiros de entrada e saída e texto de saída na consola na seguinte lista:

- Argumento do programa para executar este módulo: `-m c`
- Ficheiros de entrada: o ficheiro original ou o ficheiro do tipo `rle` se foi utilizada o mecanismo de compressão RLE; implicitamente o ficheiro `cod` correspondente também será um ficheiro de entrada.
- Texto de saída na consola:
  [identificação do(s) autor(es), curso, data, etc.]
  [identificação do módulo]
  [número de blocos]
  [lista dos tamanhos antes/depois (em bytes) e das respetivas taxas de compressão para cada bloco]
  [taxa de compressão global]
  [tempo de execução do módulo em milissegundos]
  [ficheiro gerado]
  De notar que a contagem do tempo de execução deve terminar antes de o programa fazer a escrita na consola destas mensagens.
- Ficheiros de saída: um ficheiro com o a sequência binária final da serialização de todos os códigos correspondentes aos símbolos do ficheiro de entrada, feita bloco a bloco.
- Terminação dos ficheiros de saída:
  `shaf` (ficheiro com sequência binária resultante da codificação)

Exemplo dum comando para executar este módulo sobre o ficheiro de entrada **exemplo.txt.rle** (implicitamente, o ficheiro **cod** respetivo, i.e. **exemplo.txt.rle.cod**, também será ficheiro de entrada):

```
Shell> shafa exemplo.txt.rle -m c
```

A sua execução deve gerar um ficheiro de saída **exemplo.txt.rle.shaf** e, por exemplo, o seguinte texto na consola:

```
John Doe, a12234, MIEI/CD, 1-jan-2021
Módulo: c (codificação dum ficheiro de símbolos)
Número de blocos: 2
Tamanho antes/depois & taxa de compressão (bloco 1): 57444/38962
Tamanho antes/depois & taxa de compressão (bloco 2): 1620/1210
Taxa de compressão global: 31%
Tempo de execução do módulo (milissegundos): 973
Ficheiro gerado: exemplo.txt.rle.shaf
```

D. Módulo para descodificação do ficheiro comprimido

O módulo para implementar esta última fase deve ter como ficheiros de entrada um ficheiro do tipo **cod** (implícito) onde estão as tabelas de códigos SF para todos os blocos e o ficheiro onde está a sequência binária resultante da codificação SF da fase anterior (explicitamente através do nome do ficheiro), ou seja, um ficheiro do tipo **shaf**. O módulo deve gerar como saída um ficheiro do mesmo tipo do ficheiro original de entrada para o módulo da primeira fase. Se tiver sido usada compressão RLE previamente, depois da descodificação SF em cada bloco, o módulo terá ainda que implementar a descodificação RLE para poder gerar um ficheiro igual ao original.

No caso de ter sido usada compressão RLE previamente, o utilizador pode forçar (através da opção **-d s**) a utilização apenas da descodificação SF sobre o ficheiro **shaf** e assim obter apenas um ficheiro do tipo **rle**.

Opcionalmente, se o argumento **-d -r** for usado, o módulo pode ter como ficheiros de entrada um ficheiro do tipo **freq** (implícito, para obter a informação do número e tamanho dos blocos processados) e o correspondente ficheiro onde está a sequência de símbolos resultante da codificação RLE da primeira fase (explícito através do nome do ficheiro), ou seja, um ficheiro do tipo **rle**. Note-se que não é estritamente necessário o acesso a um ficheiro **freq** para se fazer a descodificação RLE. Consegue-se fazer a descompressão RLE sem qualquer informação do número e tamanho dos blocos usados na compressão RLE da fase A. Nesse caso, em que não se está dependente de mais ficheiro nenhum que não o ficheiro do tipo **rle**, o tamanho dos blocos para fazer a descompressão RLE é decidido pelo próprio módulo f (64Kbytes) ou pelo utilizador se for implementada a opção **-b** neste módulo, tal como é no módulo f.

Mais uma vez, no final da execução o módulo deve imprimir na consola o texto de saída com informações sobre a dita execução.

Descodificar a sequência comprimida de bits

O principal objetivo desta fase é construir um módulo fazer a descodificação SF e gerar o ficheiro de símbolos que deu diretamente origem à sequência de códigos binários no ficheiro **shaf**: um ficheiro **rle** ou um ficheiro original (se a compressão RLE não foi utilizada). Adicionalmente, se houve compressão prévia RLE, então o módulo deve também implementar a descodificação RLE ao resultado da descodificação SF para se obter o ficheiro original.

A primeira funcionalidade de descodificação SF é obtida analisando consecutivamente a sequência de bits do ficheiro **shaf**, identificando as sequências encontradas como sendo iguais aos códigos SF e substituindo-as pelos símbolos correspondentes. Esta operação deve ser executada em memória, bloco a bloco.

Através da análise do ficheiro de entrada do tipo **cod**, o módulo consegue determinar se foi utilizada compressão RLE previamente à codificação SF. Se for esse o caso, então, também tem de se aplicar a descodificação RLE ao bloco de símbolos resultante da descodificação SF.

No final do processamento completo de cada bloco o resultado deve ser guardado no ficheiro de símbolos de saída até todos os blocos serem

processados. Obtém-se assim um ficheiro igual ao ficheiro original.

Otimização da descodificação binária SF

Na descompressão, para mais rapidamente se identificar o símbolo equivalente a uma determinada sequência de bits que vão sendo analisados, pode utilizar-se uma procura em árvores binárias, em que cada ramo identifica um código específico e os ramos mais perto da raiz têm os códigos mais curtos (menos bits). Este tipo de técnica, simples de programar, também é muito usado na descodificação Huffman.

Uma abordagem alternativa, ainda mais eficiente, mas mais complexa, é seguir a estratégia equivalente à codificação binária SF por matrizes de bytes.

Fica a cargo dos alunos decidir o tipo de estratégia, otimizada ou não. Em qualquer dos casos, os alunos só devem implementar algoritmos que percebam inteiramente.

Opções e ficheiros deste módulo

Pode resumir-se o conjunto de opções, ficheiros de entrada e saída e texto de saída na consola na seguinte lista:

- Argumento do programa para executar este módulo: **-m d**
- Opção para definir apenas a descompressão SF ou RLE: [**-d s|r**]
  No caso desta opção estar presente como argumento da ferramenta, então o processamento só deve contemplar a descodificação SF (opção **-d s**) para gerar um único ficheiro do tipo **rle** a partir dum ficheiro **shaf**, ou a descodificação RLE (opção **-d r**) para gerar um ficheiro original a partir dum ficheiro **rle** (neste caso o ficheiro de entrada não é do tipo **shaf** mas sim do tipo **rle**).
- Ficheiros de entrada: o ficheiro do tipo **shaf** e, implicitamente, o ficheiro **cod** correspondente; com a opção **-d r**, os ficheiros de entrada são o ficheiro **rle** e o ficheiro **freq** correspondente (este tipo de ficheiro é necessário para se saber o número e tamanho dos blocos de processamento).
- Texto de saída na consola:
  [identificação do(s) autor(es), curso, data, etc.]
  [identificação do módulo]
  [número de blocos]
  [lista dos tamanhos dos blocos antes/depois (em bytes) do ficheiro gerado]
  [tempo de execução do módulo em milissegundos]
  [ficheiro gerado]
  De notar que a contagem do tempo de execução deve terminar antes de o programa fazer a escrita na consola destas mensagens.
- Ficheiros de saída: um ficheiro de símbolos igual ao ficheiro original ou um ficheiro de símbolos do tipo **rle** (opção **-d -s**).
- Terminação dos ficheiros de saída:
  rle (ficheiro com sequência de símbolos resultante da descodificação SF do ficheiro **shaf**, caso a opção **-d -s** tenha sido usada)

Exemplo dum comando para executar este módulo sobre o ficheiro de entrada **exemplo.txt.rle.shaf** (implicitamente, o ficheiro **cod** respetivo, i.e. **exemplo.txt.rle.cod**, também será ficheiro de entrada):

```
Shell> shafa exemplo.txt.rle.shaf -m d
```

A sua execução deve gerar um ficheiro de saída **exemplo.txt** e, por exemplo, o seguinte texto na consola:

```
John Doe, a12234, MIEI/CD, 1-jan-2021
Módulo: d (descodificação dum ficheiro shaf)
Número de blocos: 2
Tamanho antes/depois do ficheiro gerado (bloco 1): 38962/65536
Tamanho antes/depois do ficheiro gerado (bloco 2): 1210/2013
Tempo de execução do módulo (milissegundos): 973
Ficheiro gerado: exemplo.txt
```

Otimização por execução paralela

A utilização de processamento em blocos independentes que foi definida na especificação dos módulos da ferramenta permite a eventual execução paralela do processamento de grupos de blocos, caso o sistema operativo implemente um adequado aproveitamento dos vários CPUs e/ou multi-núcleos presentes na máquina. O aproveitamento da otimização da execução em paralelo só pode ser feito se as partes do código que processam um bloco for executada em diferentes *threads*.

Por exemplo, em todos os módulos podem criar-se até **T** *threads* (dependendo do número de blocos disponíveis para processamento) para executar o processamento em paralelo de **T** blocos de dados. O sistema operativo irá adaptar o número de *threads* a correr em paralelo ou concorrentemente dependendo dos CPUs/núcleos disponíveis na máquina, por isso, não é preciso fazer **T** menor que o número de CPS/núcleos da máquina.

De notar que a implementação de várias *threads* para eventual processamento dos blocos em paralelo obriga a uma gestão atenta do processo de leitura e escrita dos ficheiros de entrada e saída:

- Na leitura dos dados de entrada dos ficheiros originais, dos ficheiros `rle` ou dos ficheiros `shaf`, deve ser garantida uma leitura sequencial (os dados dos outros tipos de ficheiros podem ser lidos duma única vez para memória). Os blocos a serem processados devem ser lidos sequencialmente para garantir que todos os bytes têm uma posição relativa em memória igual à posição relativa nos ficheiros. São depois entregues às *threads* para processamento paralelo. Esta estratégia pode ser implementada de duas formas alternativas. Ou a execução das threads só se inicia quando todos os blocos já tiverem sido lidos para memória (ver Figura 7) ou a entrega dos dados às *threads* pode ser feito à medida que os blocos vão sendo lidos, uma *thread* de cada vez (ver Figura 8) e, nesse caso, as *threads* não se vão iniciar todas ao mesmo tempo.

- Na escrita dos resultados o problema os cuidados a ter são semelhantes. Cada *thread* processa os dados e coloca o resultado num local temporário em memória e é preciso garantir que os dados são escritos no ficheiro de saída pela ordem correta. Mais uma vez, existem duas abordagens possíveis. Ou a escrita dos resultados do processamento dos blocos só se inicia quando todos os blocos acabarem de ser processados (ver Figura 7) ou os resultados do processamento vão sendo escritos à medida que vão acabando (ver Figura 8). No primeiro caso é fácil garantir a ordem correta da escrita dos blocos de resultados no ficheiro de saída pois a escrita já é feita fora das *threads*. No segundo caso, em que os blocos de resultados são gravados ainda nas respetivas *threads*, tem de forçar-se a escrita dos blocos duma forma sequencial, sem paralelismo e na ordem correta (o que, apesar de tudo, é de fácil implementação utilizando uma variável global de monitorização).

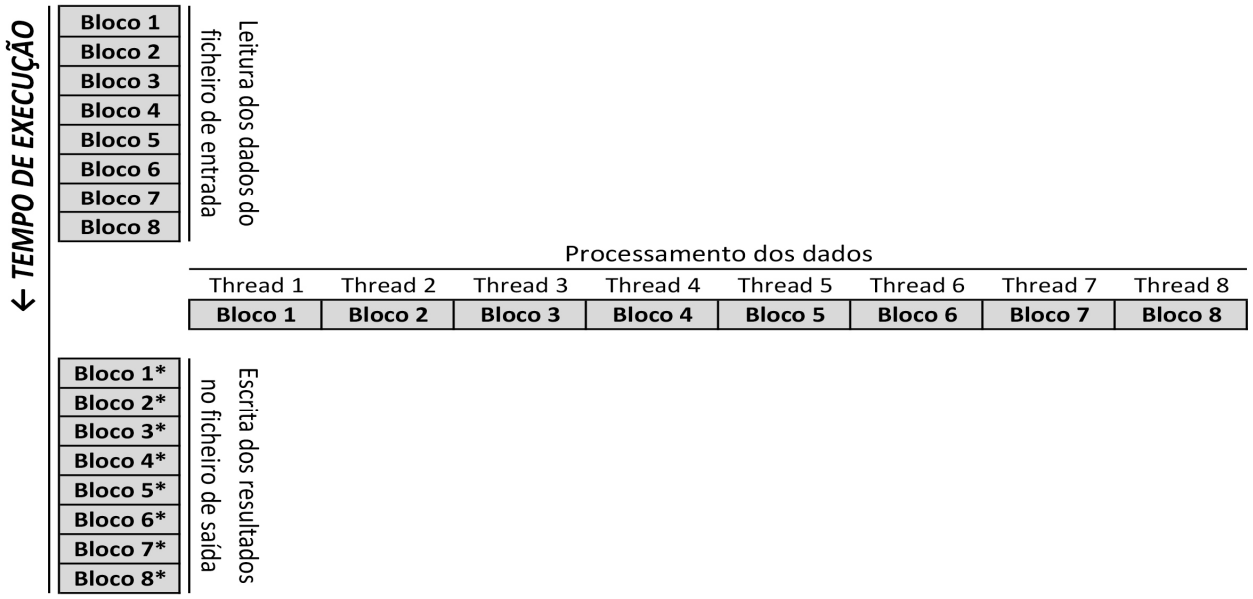

Figura 7: Gravação dos resultados fora das *threads*.

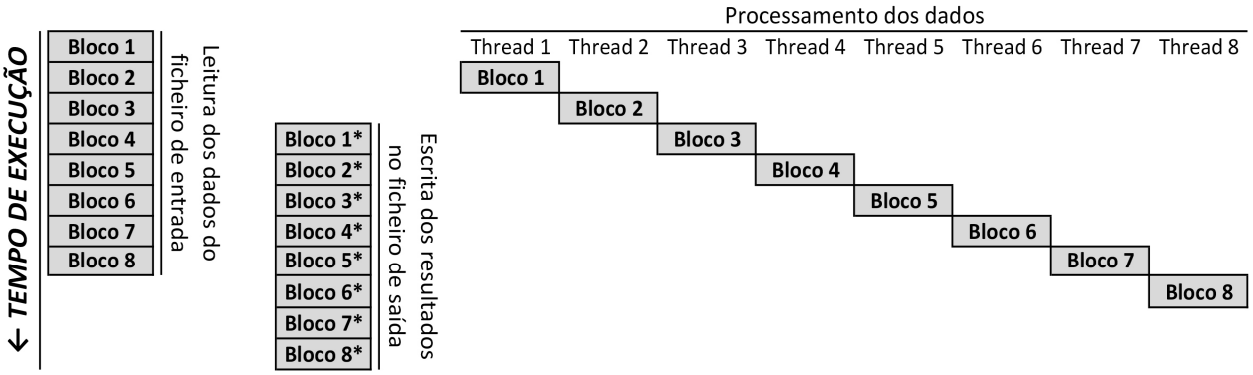

Figura 8: Gravação dos resultados dentro das *threads*.

Relatório e outras recomendações

A ferramenta/módulos criados devem ser testados com ficheiros de teste de vários tamanhos e de vários tipos (ficheiros de texto, imagem, vídeo, PDF, etc.). Os docentes também fornecerão alguns ficheiros.

O código desenvolvido pode ter embutido um modo de funcionamento em *debug* (ou *verbose*) em que imprima informações relevantes durante a sua execução e que possam ser úteis na análise do seu comportamento em situações de erro ou de funcionalmente incorretas.

O código deve utilizar apenas funções normalizadas da linguagem C da norma 217 ISO (STD 17) e ou funções desenvolvidas pelo grupo de trabalho. Não use APIs, funções ou excertos de código de terceiros.

O código deve ser claro e usar convenções de nomeação de variáveis, tipos, funções e constantes. O código deve ser estruturado duma forma o mais modular possível, sem complexidades desnecessárias, e permitir reutilização sempre que possível.

A qualidade e correção do código não se mede pelo seu tamanho.

Os alunos devem documentar/explicar o código criado através de comentários no nas secções de código mais relevantes e no relatório. Todos os ficheiros do código devem ter um cabeçalho com a informação relevante que identifique os seus autores e explique as principais funções criadas, tipos de dados e variáveis usadas, etc.

O relatório deve incluir:

- Na primeira página, o título do trabalho e a identificação dos autores (incluindo fotografia), universidade, curso, unidade curricular e data de entrega;
- Um índice do conteúdo;
- Na primeira secção, a descrição da organização do grupo em torno do trabalho desenvolvido, quem contribuiu para o quê;
- Na segunda secção, a discussão das estratégias escolhidas, as opções tomadas e os mecanismos adotados, incluindo eventuais otimizações;
- Na terceira secção, a explicação e análise critica das principais funções implementadas no código, os seus principais méritos e a suas limitações mais importantes;
- Na quarta secção, uma tabela a sumariar uma sequência de resultados da execução dos módulos a vários ficheiros exemplificativos;
- Uma secção de conclusões que inclua uma eventual discussão sobre o que gostava de ter feito melhor ou de ter acrescentado e não conseguiu;
- Uma lista de eventuais referências bibliográficas, artigos científicos ou recursos informais na web e que tenham sido úteis.

O relatório é para ser avaliado pelos docentes por isso não inclua informação genérica e irrelevante que os docentes já conheçam. Tente ser conciso e claro. Sempre que incluir uma afirmação importante no contexto do relatório e que seja de autoria de terceiros, ou que seja baseada diretamente em afirmações de terceiros ou concluída de informação retirada de recursos alheios, deve referenciar corretamente essas autorias ou

proveniências e acrescenta-las na lista das referências.

O relatório pode incluir algumas partes relevantes do código quando estas ajudam às análises e justificações apresentadas. Não inclua código desnecessário no texto do relatório. Sempre que possível, comente antes o próprio código.

Antes da entrega do material para avaliação, e o mais cedo possível, todos os elementos do grupo devem registar-se no grupo respetivo no *black-board* da UC. A entrega do material deve ser feita recorrendo ao upload do material na pasta do grupo no *black-board*.

Na entrega, inclua apenas dois ficheiros: um ficheiro PDF com o relatório e um ficheiro zip com todos os ficheiros do código do projeto dentro duma diretoria com o seguinte nome CD2021-GXX, em que XX é o número do grupo, como por exemplo, CD2021-G02. O ficheiro principal do projeto e o ficheiro .c principal deve chamar-se **shafa**.

Por fim, é recomendável que, durante a defesa do trabalho, tente responder honesta e concisamente apenas às questões colocadas por forma a que as sessões de apresentação não se arrastem muito para além dos 60 minutos.