Universidade do Minho
Mestrado Integrado em Engenharia Informática

Relatório do projeto SHAFA

Comunicação de Dados | 4 de Janeiro, 2021

Módulo A

João Mateus Costa Coelho
A93255

Diogo Miguel da Silva Araújo
A93313

Módulo B

Ana Filipa Ribeiro Murta
a93284

Ana Paula Oliveira Henriques
a93268

Módulo C

Diogo da Costa e Silva Lima Rebelo
a93180

Joel Costa Araújo
a76603

Módulo D

Luís Manuel Peixoto Silva
a93293

Gonçalo da Ponte Carvalho
a93260

Índice

Introdução

O presente relatório tem como objetivo apresentar e informar sobre os diversos passos dados ao longo da realização do projeto proposto pelos docentes da disciplina Comunicação de Dados. O objetivo deste projeto foi criar um executável capaz de comprimir (e por consequência descomprimir) ficheiros através de codificação Shannon-Fano.

Para começar, este foi realizado por um grupo de 8 pessoas, divididas em 4 subgrupos de pares, tendo sido atribuído a cada par um dos módulos do projeto.

O módulo A/f foi atribuído aos alunos João Mateus Costa Coelho e Diogo Miguel da Silva Araújo, onde, entre si, dividiram tarefas confiando, essencialmente, a compressão RLE com o Mateus Coelho e o cálculo das frequências com o Diogo Araújo. Entre ambas as tarefas houve, no entanto, entreajuda e partilha de ideias e métodos de resolução diferentes para cada parte.

O módulo B/t foi atribuído às alunas Ana Filipa Ribeiro Murta e Ana Paula Oliveira Henriques, sendo, em geral, tudo feito em conjunto.

O módulo C foi atribuído aos alunos Diogo da Costa e Silva Lima Rebelo e Joel Costa Araújo sendo, mais uma vez, tudo feito em conjunto.

O módulo D foi atribuído aos alunos Luís Manuel Peixoto Silva e Gonçalo da Ponte Carvalho, sendo divididas as tarefas pelos 2 colegas. A descompressão Shaf ficou com o Luís Silva e a descompressão RLE ficou com o Gonçalo, havendo sempre comunicação de problemas e ajuda entre ambos.

Estratégias

- ## Módulo A:

Para a compressão RLE, realizada no primeiro módulo deste projeto, a ideia principal, que foi levada em frente, consiste numa divisão, em diversos blocos do tamanho pretendido, do ficheiro original, de seguida a sua análise em termos de bytes repetidos de seguida, para a posterior reorganização no método pretendido. Para tal, foram utilizadas duas funções, auxiliares a esta parte, percorrendo todos os elementos de cada bloco (apenas um bloco é percorrido de cada vez, sendo também apenas criado um de cada vez) e guardando tanto o resultado final da compressão, como o seu novo tamanho. Para tal, sentiu-se a necessidade de criar novos tipos de dados, capazes de armazenar, de forma organizada e no seu todo, os resultados deste processo.

Por sua vez, para o cálculo das frequências foram utilizadas funções auxiliares mais simples para posterior utilização. A cada bloco criado foi atribuído um array de 255 inteiros, onde se armazenou a quantidade de vezes que cada elemento aparecia no respetivo bloco, por incrementação unitária. Deste modo, foi necessária apenas a implementação desta ideia para o número total de blocos que havia para analisar.

- ## Módulo B:

A estratégia aplicada no módulo B foi primeiramente elaborar as funções relativas à tabela dos códigos Shannon-Fano. Inicialmente fizemos uma função que ordenasse decrescentemente a lista de frequências. Em seguida, utilizámos o algoritmo fornecido pelo professor, que calcula as melhores divisões e os códigos Shannon-Fano.

Posteriormente, partimos para o algoritmo que recebe o ficheiro .freq e devolve o ficheiro .cod. A estratégia, neste caso, foi utilizar um buffer para guardar o conteúdo do ficheiro recebido. Seguidamente, lemos o conteúdo entre dois arrobas e utilizámos a mesma estratégia para ir buscar as frequências entre os pontos e vírgulas, guardando-as num array. Por último, criámos o ficheiro .cod com os códigos das frequências calculadas.

- ## Módulo C:

Tínhamos como estratégia inicial tratar os ficheiros bloco a bloco, guardando os caracteres do ficheiro em um array e dividindo esse mesmo bloco a bloco. Ao longo do desenvolvimento do projeto acabamos por guardar num array separado o bloco que estamos a tratar e chegando ao fim colocamos nesse array o bloco seguinte. Uma das nossas estratégias foi o uso de matrizes para armazenar os valores binários na posição certa e o seu valor original.

- ## Módulo D:

Para o módulo D na leitura do ficheiro .cod é utilizado um mecanismo de árvores binárias. Estas árvores guardam a posição do código binário no inteiro code e têm dois apontadores: um e zero. O um e o zero vão formar um caminho até ao code que representará um número binário lido do .cod. Posteriormente essas mesmas árvores binárias são lidas, ajudando assim na descodificação SF, uma vez que se torna mais fácil quais os elementos do ficheiro cod.

Na descodificação do rle é tomada uma estratégia mais simples, cuja única otimização é a leitura do ficheiro .freq para descobrir o número de blocos e o seu respetivo tamanho, tornando mais fácil a descompressão. A estratégia passou apenas

por ler cada elemento e descobrir se é “00”, caso seja faz descompressão, caso não seja apenas imprime o valor lido no output.

Caso haja algum erro a correr o programa pode ser do tamanho limite definido no programa (10000000).

Principais Funções

- ## Módulo A:

De maneira geral, o primeiro módulo deste projeto tem duas funções principais importantes, responsáveis para divisão em blocos e posterior compressão RLE, ou não, caso não seja vantajoso, e uma outra que analisa esses blocos e calcula-lhes as suas frequências de caracteres.

Numa primeira abordagem, a compressão RLE foi feita com alguma destreza e simplicidade. Preliminarmente, após ter sido analisado o conteúdo do ficheiro original de entrada, atuou-se diretamente no mesmo para o comprimir, analisando as repetições seguidas dos mesmos caracteres e adotando uma imediata conversão para o formato pretendido. Tanto o resultado final, como o tamanho do mesmo, são guardados num novo tipo de dados, ao qual chamamos BLOCO_T, para lhes ter fácil acesso aquando da escrita dos mesmos. De seguida, no processo final, são tidas em consideração as variáveis possíveis, isto é, se é vantajoso fazer a compressão, se há blocos com tamanho suficiente ou se é necessária a sua junção e se esta está a ser forçada pelo utilizador. No final, aplica-se o método, anteriormente explícito, e guardam-se, num outro tipo de dados criados, CARACT_FICHEIRO_COMP, os pormenores desta parte para posterior utilização no cálculo das frequências. O principal problema desta função, “compress”, será a sua extensão, de modo que a leitura da mesma pode tornar-se mais difícil do que as restantes e o facto de escrever as informações no terminal, visto que era pretendido que as mesmas fossem escritas fora dos intermédios, para não contabilização do tempo despendido na sua escrita. No entanto, é de notar que são atentados vários pormenores fulcrais a um bom funcionamento do programa.

Por sua vez, neste cálculo há uma função principal, “frequências”, que, através dos dados previamente guardados, é capaz de percorrer os blocos dos ficheiros pretendidos e calcular-lhes o número de vezes que aparece cada caracter, utilizando o método explícito no tema anterior. Deste modo, e findado o processo, procede à respetiva escrita num ficheiro .freq de uma forma bastante descomplicada, através de um “loop for” que procede, bloco a bloco, à criação de um array com o cálculo das frequências e a escreve logo de seguida, no ficheiro pretendido, juntamente com o tamanho dos respetivos blocos. É, portanto, de acessível e transparente interpretação, além de concisa.

- ## Módulo B:

Neste módulo as funções principais são a calculte_codes_SF e a encode.

A calculate_codes_SF, tal como o nome indica, calcula os códigos Shannon-Fano recebendo uma lista de frequências de tipo inteiro. Depois de calculados os códigos para cada frequência, estes são guardados no array codes do tipo unsigned char. Para além disto, este algoritmo utiliza a variável div para guardar o valor da melhor divisão da lista de frequências. Já as variáveis start e end limitam a lista de frequências para qual se vai calcular os códigos Shannon-Fano. O valor destas variáveis vai se alterando de modo a determinar os códigos para cada frequência, ou seja, de acordo com o div, calculado pela função calculate_best_divider, no array codes, vamos colocar o bit ‘0’ nas posições entre os índices start e div e o bit ‘1’ entre os índices div+1 e end, aplicando a função calculate_codes_SF recursivamente para estes subconjuntos.

A encode recebe o ficheiro .freq e retorna o ficheiro .cod. Inicialmente, esta função guarda o conteúdo do ficheiro num buffer em memória. Em seguida, será processado um bloco de cada vez. Posteriormente, a tática utilizada neste algoritmo é ir buscar o conteúdo entre dois arrobas utilizando a função count_char_arroba (função

fornecida pelo grupo do módulo D). Deste modo, para além desta função, a encode também utiliza esta estratégia para ir buscar as frequências e colocá-las num array, designado por listFreq, que depois irá ser passado como argumento para a função ordena e sucessivamente para a calculate_codes_SF. Por último, o array codes, calculado pela calculate_codess_SF, é utilizado para construir o ficheiro .cod. Assim sendo, encode é a função que invoca, indiretamente, todas as funções deste módulo, isto é, por exemplo a função swap é utilizada pela sort_list, que por sua vez é chamada pela encode.

Durante este trabalho, as nossas principais limitações foram perceber como iríamos ler o ficheiro .freq, de modo a ir buscar as frequências para calcular os códigos Shannon-Fano.

- ## Módulo C:

As nossas funções principais são “fPrincipal”, “conversao” e “matriz_to_string”.

Na fPrincipal vamos inicializar os ficheiros que vamos usar e os arrays que iram conter esses mesmos ficheiros. Vamos inicializar os arrays para guardar o conteúdo desses mesmos ficheiros, iniciamos o temporizador e outras funções auxiliares. É também na fPrincipal onde vamos imprimir informações uteis como taxa de compressão global.

Na conversão começamos por inicializar e alocar espaço para a matriz que conterá a sequência binaria do bloco em que se esta a trabalhar. O nosso objetivo seria conseguirmos alcançar uma string com a sequência binaria separada bit a bit (Ex [1,0,0] em vez de [10,0]).

Como estávamos a substituir um caracter específico no bloco inteiro (Ex: substituir de uma vez só todos os 'a'), achamos melhor criar uma matriz com a quantidade de linhas igual ao número de caracteres do bloco e assim substituir nas linhas especificas através da função auxiliar substitui_matriz (Ex. “aabd” → [[1,0] , [1,0] , [0] , [1,1,0]]). Deste modo foi possível, após a matriz estar completa, converter a matriz numa string com a sequencia binaria separada bit a bit, como pretendíamos, facilitando a codificação.

A função matriz_to_string tem como objetivo criar uma string cujo conteúdo é a sequência de bits de modo a futuramente facilitar o agrupamento de 8 a 8 para a conversão final. Para que tal seja possível precisamos de uma string que contenha e nos deixe aceder aos números binários 1 a 1. Deste modo a função percorre a matriz e copia o seu conteúdo para a string e no processo calcula o tamanho da string, possibilitando outros cálculos como por exemplo o tamanho dos blocos comprimidos e número necessários a adicionar se necessário.

- ## Módulo D:

O módulo D tem uma função principal, chamada decode, que depois se acaba por dividir em 2 funções: decodesf (composta por decodeCOD e decodeSHAF) e decoderle.

A abordagem na função decodeCOD passou por, numa fase inicial, começar por descobrir qual o tipo do ficheiro cod, ou seja, se este era rle ou shaf, sendo que isso foi conseguido com a ajuda da função contachars, que lia a primeira parte do ficheiro e verificava se aparecia um “R” ou um “N”. Passada esta fase, ainda com a ajuda da contachars, a função lê o número de blocos e cria uma árvore binária para cada um desses blocos, todas elas inicialmente vazias. À medida que cada bloco é lido, sempre que é encontrado um novo elemento, este é adicionado à árvore de forma binária, isto é, é lido o número e consoante a leitura, caso seja lido um 0 passa-se para árvore da esquerda, caso seja um 1 passa-se para a da direita, sendo que a cada nodo está associado o inteiro “code” que serve para identificar o elemento guardado. A maior valência desta função é o facto de tornar muito mais

simples e eficaz o trabalho de leitura da função decodeSHAF, enquanto a sua maior limitação é não estar completamente otimizada, tendo algumas variáveis que acabam por ser dispensáveis.

Na função decodeSHAF vai ser feita a descodificação SF, sendo que, inicialmente pode ter que ser feita a descodificação RLE, sendo chamada a função decodeBlocoRLE, dependendo a informação recebida na variável info. Tal como na anterior, também aqui o ficheiro vai ser descodificado por blocos, com a ajuda da função, decodeBLOCOSHAF, sendo que é aqui que são utilizadas as árvores binárias criadas pela função decodeCOD. A descodificação passa por verificar cada elemento do array value e verificar se há algum nodo em que o valor “code” seja equivalente ao elemento, sendo que isto é feito verificando o valor da variável “belongs”, que é um inteiro que ou indica a posição desse elemento, sendo utilizada a função procura para colocar em belongs o valor da posição, ou então apresenta o valor “-1” que significa que esse elemento não pertence. Sempre que é encontrado um novo elemento este é adicionado ao array “final”. A maior vantagem desta função é o seu nível de otimização, que é bastante superior ao das outras duas, enquanto a sua maior limitação é a sua grande dependência das outras funções.

A função decoderle , tal como o nome indica, serviu para fazer a descompressão RLE. A estratégia utilizada passou por ler o número de blocos do ficheiro a descodificar, com a ajuda da função calculaRLETam, que lia o ficheiro freq e indicava o número de blocos e o respetivo tamanho. Já com a informação acerca dos blocos, a estratégia de descodificação passou apenas por fazer a leitura binária do ficheiro. Sempre que fosse encontrado o valor “00” é chamada a função seqRLE para “descomprimir”, sendo que esta função guardava os 2 valores seguintes, o primeiro para saber o símbolo e o segundo para saber o número de vezes que o símbolo teria que ser repetido. A maior vantagem desta função é a sua simplicidade, sendo fácil de entender. A sua maior limitação também acaba por ser essa mesma simplicidade, uma vez que poderiam ser implementadas melhorias que acabariam por tornar a função mais eficiente.

Resultados

- **Módulo A:**

Nome do ficheiro	Tamanho do ficheiro original (bytes)	Comando usado	Tempo decorrido (milisegundos)	Tamanho do ficheiro comprimido (bytes)	Taxa de compressão
aaa.txt	3000	shafa aaa.txt -m f	3	1422	52.6 %
Shakespeare.txt	5776694	shafa Shakespeare.txt -m f -b K -c r	764	5672284	1.8 %
bbb.zip	750030201	shafa bbb.zip -m f -b m -c r	122994	753797532	-0.5 %
random_.jpeg	2505760	shafa random_.jpeg -m f -b K -c r	362	2546615	-1.6 %
teste.mp4	171085	shafa teste.mp4 -m f -b K -c r	27	148479	13.2 %

- **Módulo B:**

Nome do ficheiro	Tamanho do ficheiro original (bytes)	Comandos usados	Tempo executado (milisegundos)
aaa.txt.freq	556	.\shafa.exe aaa.txt.freq -m t	3
aaa.txt.rle.freq	592	.\shafa.exe aaa.txt.rle.freq -m t	16
bbb.zip.freq	138 940	.\shafa.exe bbb.zip.freq -m t	144
bbb.zip.rle.freq	138 941	.\shafa.exe bbb.zip.rle.freq -m t	157
Shakespeare.txt.freq	458 082	.\shafa.exe Shakespeare.txt.freq -m t	23
Shakespeare.txt.rle.freq	458 082	.\shafa.exe Shakespeare.txt.rle.freq -m t	31

- **Módulo C:**

Nome do ficheiro	Tamanho do ficheiro original (bytes)	Comando usado	Tempo decorrido (milissegundos)	Tamanho do ficheiro comprimido (bytes)	Taxa de compressão
aaa.txt	3000	shafa aaa.txt -m c	2	671	22,37%
aaa.txt.rle	1422	shafa aaa.txt.rle -m c	3	473	33,26%
Shakespeare.txt	5776694	shafa Shakespeare.txt -m c	4820	3505734	60,69%
Shakespeare.txt.rle	5672284	shafa Shakespeare.txt.rle -m c	7115	3542227	62,45%

- **Módulo D:**

Nome do ficheiro	Tamanho do ficheiro original (bytes)	Comando usado	Tempo de decode (milissegundos)
aaa.txt.rle.shaf	493	shafa aaa.txt.rle.shaf -m d	1.56
bbb.zip.shaf	752455728	shafa bbb.zip.shaf -m d	669102.00
bbb.zip.rle	753797532	shafa bbb.zip.rle -m d	5507.66

Conclusões

Concluímos este relatório com uma breve visão geral sobre todos os módulos, referindo aspetos onde poderíamos ter melhorado os mesmos. Gostaria ainda de acrescentar que cada par criou e expôs neste relatório os seus módulos independentemente, (tendo havido, quando necessário, ajuda entre todos os constituintes do grupo) uma vez que não vimos sentido em falar sobre um módulo onde não participamos ativamente.

No caso do módulo A|f, teria sido de melhor aproveitamento uma divisão de funções em outras mais simples, para que não se tornasse tão difícil de acompanhar a sua leitura. Em termos daquilo que gostaríamos de fazer, que não conseguimos, foi trabalhar em algum processo de otimização para este módulo. Seguimos com as ideias principais que foram surgindo e, devido ao contínuo foco nas mesmas, não conseguimos, agora numa fase mais final, pensar e adaptar o nosso código a algo que, efetivamente, tivesse resultados mais rápidos. Pensamos, assim, que provavelmente haveria forma de arranjar melhorias nesse aspeto, mas não o conseguimos.

No módulo B|t, um dos aspetos que gostaríamos de ter melhorado é o facto de termos um código pouco simplificado. Para além disto, um erro que queríamos ter corrigido a tempo é a má leitura da frequência “0” (zero), uma vez que, em vez de colocar o zero na listFreq, coloca o “48”. Ou seja, o “0” é lido como “48”, segundo a tabela ASCII.

No módulo C um dos problemas que tivemos durante o desenvolvimento deste projeto foi a maneira que iriamos escrever num array os valores binários, visto que iriamos precisar de os agrupar 8 a 8. Nisto reparamos num problema, o array teria os valores binários já agrupados, como por exemplo [10;11;110;0;10]. Isto tornaria difícil o agrupamento 8 a 8 visto que seria difícil saber quantos números binários estariam em cada posição do array. Tivemos algumas dificuldades a utilizar a função fsize fornecida pelo equipa de docentes o que nos levou a usar apenas uma pequena parte dessa função para determinar o tamanho de um ficheiro. O resto da função que não conseguimos por a funcionar no nosso código, acabamos por fazer as nossas próprias funções. Contudo temos consciência que o código poderia estar mais bem organizado em certas partes e gostaríamos de ter tido tempo para implementar algumas otimizações no código.

No módulo D ficamos satisfeitos com o resultado globa, apesar de acharmos que poderia haver melhorias em 2 pontos distintos, sendo eles o comando “b”, que gostaríamos de ter conseguido acrescentar e que não foi possível, e a implementação de threads, que, não sendo tão importante, também sentimos que seria uma boa adição a este módulo. Também há que referir que há um pequeno problema aquando da execução do programa em Linux na transformação dos ficheiros .rle.shaf para .rle.

Referências Bibliográficas

https://www.geeksforgeeks.org/

https://www.programiz.com/

https://www.tutorialspoint.com/

https://stackoverflow.com/

https://www.clubedohardware.com.br/topic/1031279-resolvido-medir-tempo-de-execu%C3%A7%C3%A3o-em-c/