Relatório CD 2021 - G22

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Comunicação de Dados / 3 de Janeiro 2021

Figura 1: Módulo A Júlio Beites Gonçalves

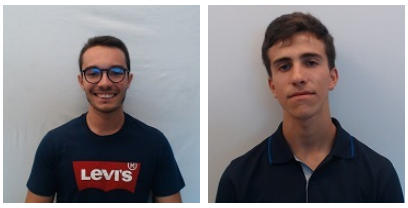

Figura 2: Módulo B Norberto Alexandre Nunes/ João Duarte Laranjo Cerquido

Figura 3: Módulo C Daniel Constantino Faria/ João Augusto Moreira

Figura 4: Módulo D João Luís Santos/ Lídia Anaís Coelho Sousa

Conteúdo

1 Organização do grupo

O grupo ficou dividido em 4 subgrupos, todos com 2 elementos à exceção do subgrupo encarregue pelo módulo A apenas composto por 1 elemento.
Ficaram assim encarregues pelos módulos:

Módulo A: Júlio Beites Gonçalves

Módulo B: João Duarte Laranjo Cerquido, Norberto Alexandre Nunes

Módulo C: Daniel Constantino Faria, João Augusto Moreira

Módulo D: João Luis Santos, Lídia Anais Coelho Sousa

Relatório(Partes Comuns) : João Augusto Moreira, Lídia Anaís Coelho Sousa

Repositório e Estruturas : João Duarte Laranjo Cerquido, João Luis Santos

Função Main e Pontos Gerais : Júlio Beites Gonçalves, Daniel Constantino Faria

2 Estratégias usadas

2.1 Módulo A

Neste módulo, após o estudo das várias funcionalidades a implementar nesta fase do programa comecei por trabalhar com arrays dinâmicos. Estes seriam utilizados como buffers para armazenar em memória blocos de texto lidos do ficheiro.

Após várias atualizações do enunciado e consequentemente estudo do trabalho indirectamente e directamente, optei também por utilizar matrizes com alocação de memória.

O ficheiro original é dividido por blocos que são ordenadamente lidos para memória, para facilitar a execução. Em memória, é efetuada a compressão e respectivo calculo das frequências do ficheiro original. O calculo das frequências sobre o ficheiro .rle é feito posteriormente, sendo analisadas as suas ocorrências através do ficheiro .rle já criado.

2.2 Módulo B

A estratégia relativa a este módulo baseia-se em ler todas as frequências de todos os blocos do ficheiro .freq para uma matriz. Esta matriz é posteriormente convertida numa lista ligada para tornar possíveis ordenações (como por ordem decrescente ou lexicográfica) mantendo a informação de qual símbolo corresponde a qual frequência. Mais tarde nesta mesma estrutura é colocado o código Shannon-Fannon correspondente a cada símbolo (armazenado também na forma de uma lista ligada). Finalmente é só preciso percorrer esta lista ligada e escrever no ficheiro .cod os códigos com a sintaxe estabelecida.

2.3 Módulo C

Neste módulo, com base nas recomendações e propostas do enunciado do guião do projeto, decidimos trabalhar em torno de matrizes e acessos a memória.

Então, como sugerido, colocámos em buffers, ou seja, arrays, a informação dos ficheiros de entrada e, posteriormente, guardar em matrizes uma vez que resulta num acesso mais facilitado e ordenado. Utilizando este método podemos modificar e manusear esta mesma informação em memória e passá-la para o ficheiro de saída.

2.4 Módulo D

O objetivo deste módulo é a descompressão dum ficheiro para a sua versão original. Como tal recebemos um ficheiro .cod e um ficheiro .shaf e, no caso de ter sido usado a compressão RLE, teremos ainda de implementar a descodificaçã0 RLE. Tendo em conta o conhecimento adquirido de unidades curriculares como LI2 e PI, optamos por trabalhar com listas ligadas, matrizes e árvores binárias por considerarmos que seria a aproximação mais simples e mais eficiente dentro das nossas capacidades. Desta forma, todos os ficheiros que recebíamos colocávamos num buffer para trabalhar com acessos à memória. E de seguida trabalhávamos com eles em matriz, transformando-os posteriormente em listas ligadas e árvores dependendo do objetivo.

3 Explicação e análise critica do código

3.1 Módulo A

O meu módulo divide-se em duas funções principais, a função normal_compressão e a função forca_compressão, diferenciando apenas se é executado o comando -c r que força o programa a executar a compressão RLE sobre o ficheiro original independentemente da compressão obtida.

```
v = criarBufferArray(file_name, tamanho, bloco);           // Copia um bloco para memória
j = criarBufferMatriz(BlocosLength);                       // Cria uma matriz
j = frequenciaCalculo(v, tamanho, bloco, j);    // Guarda em memória as frequencias de cada simbolo para cada bloco

caracterRLE = compressaoRLE(file_name, v, tamanho, bloco); // Comprime e escreve no ficheiro .rle bloco a bloco && retorna o numero de caracteres final.
taxa = taxaCompressao(caracterRLE, v);                     // Calcula a taxa de compressão
blocosRLE[RleB] = caracterRLE;                             // Guarda o tamanho de cada bloco depois de comprimido
RleB++;                                                    // Atualiza o bloco
bloco++;                                                   // Atualiza o bloco

if (taxa >= 5 && BlocosLength > 1) // Condição de teste
{
    while (bloco < BlocosLength - 1) // Condição para saltar no ultimo bloco
    {
        v = criarBufferArray(file_name, size, bloco);              // Copia um bloco para memória
        j = frequenciaCalculo(v, size, bloco, j);                      // Guarda em memória as frequencias de cada simbolo para cada bloco
        blocosRLE[RleB] = compressaoRLE(file_name, v, size, bloco); // Guarda o tamanho de cada bloco depois de comprimido
        RleB++;                                                    // Atualiza o bloco
        bloco++;                                                   // Atualiza o bloco
        free(v);                                                   // Liberta espaço alocado na memória
    }

    int a = tamanhoUltimoBloco(FileLength, size);           // Calcula o tamanho do ultimo blocos
    v = criarBufferArrayFinal(file_name, size, a, bloco);    // Copia o ultimo bloco para memória
    j = frequenciaCalculo(v, size, bloco, j);                    // Guarda em memória as frequencias de cada simbolo para o ultimo bloco
    blocosRLE[RleB] = compressaoRLE(file_name, v, a, bloco); // Guarda o tamanho do último bloco depois de comprimido
}
else if (taxa < 5 && BlocosLength > 1)
{
    while (bloco < BlocosLength - 1) // Condição para saltar no ultimo bloco
    {
        v = criarBufferArray(file_name, size, bloco); // Copia um bloco para memória
        j = frequenciaCalculo(v, size, bloco, j);          // Guarda em memória as frequencias de cada simbolo para cada bloco
        bloco++;                                      // Atualiza o bloco
        free(v);                                      // Liberta espaço alocado na memória
    }

    int a = tamanhoUltimoBloco(FileLength, size);        // Calcula o tamanho do ultimo blocos
    v = criarBufferArrayFinal(file_name, size, a, bloco); // Copia o ultimo bloco para memória
    j = frequenciaCalculo(v,a, bloco, j);                  // Guarda em memória as frequencias de cada simbolo para o ultimo bloco
}
```

Figura 5: Código normal_compressão

A lógica seguida sempre pelo meu programa é a cópia de um bloco do ficheiro para memória, seguido do cálculo das frequências sobre o ficheiro original. De seguida efetua a compressão RLE sobre o bloco em memória e escreve diretamente no ficheiro de saida .rle.

Adicionalmente, foram criadas funções idênticas mas que se limitam apenas ao processamento do último bloco do ficheiro uma vez que este vai apresentar caracteristicas diferentes dos restantes.

Ao longo da execução desta função são também guardadas informações que virão a ser utilizadas depois tais como: o tamanho de cada bloco após a sua compressão, a taxa de compressão, o tamanho do ficheiro original, numero de blocos a serem processados, entre outros...

O cálculo das frequências sobre o ficheiro .rle apenas é efetuada perto do final da execução do programa. Este copia cada bloco já compresso para memória novamente, onde efetua o cálculo das suas frequências. Posteriormente, vai criar um ficheiro com a extensão .freq onde vai inscrever a informação referente ao ficheiro .rle .

Por último, é invocada a função imprimeTerminal com a finalidade de reproduzir no terminal um texto informativo referente à execução do módulo A.

```c
// Função executada através do comando "-c r"
int forca_compressao(char *file_name, int size)
{
    unsigned char *v;
    int **j;
    int bloco = 0;
    int caracterRLE = 0;
    int RleB = 0;
    int taxa;
    unsigned long FileLength = tamanhoFicheiro(file_name);           // Calcula o tamanho do ficheiro .txt
    int BlocosLength = quantidadeBlocos(FileLength, size); // Calcula em quantos blocos vai ser divido o ficheiro original
    int *blocosRLE = malloc(sizeof(int) * BlocosLength);   // Guarda o tamanho de cada bloco depois de comprimido

    if (BlocosLength > 1) // Condição de teste
    {
        j = criarBufferMatriz(BlocosLength);                    // Cria uma matriz
        while (bloco < BlocosLength - 1) // Condição para saltar no ultimo bloco
        {
            v = criarBufferArray(file_name, size, bloco);           // Copia um bloco para memória
            j = frequenciaCalculo(v, size, bloco, j);                   // Guarda em memória as frequencias de cada simbolo para cada bloco
            caracterRLE = compressaoRLE(file_name, v, size, bloco); // Comprime e escreve no ficheiro .rle bloco a bloco && retorna o numero de caracteres final.

            if (bloco == 0) // Calcula a taxa de compressão para o primeiro bloco
                taxa = taxaCompressao(caracterRLE, v);

            blocosRLE[RleB] = caracterRLE; // Guarda o tamanho de cada bloco depois de comprimido
            RleB++;                        // Atualiza o bloco
            bloco++;                       // Atualiza o bloco
            free(v);                       // Liberta espaço alocado na memória
        }

        int a = tamanhoUltimoBloco(FileLength, size);           // Calcula o tamanho do ultimo blocos
        v = criarBufferArrayFinal(file_name, size, a, bloco);    // Copia o ultimo bloco para memória
        j = frequenciaCalculo(v,a, bloco, j);                     // Guarda em memória as frequencias de cada simbolo para o ultimo bloco
        blocosRLE[RleB] = compressaoRLE(file_name, v, a, bloco); // Guarda o tamanho do último bloco depois de comprimido
    }
    else // Caso o ficheiro seja todo processado num unico bloco
    {
        int tamanho;
        if(FileLength > size) tamanho = size;
        else tamanho = FileLength;

        v = criarBufferArray(file_name, tamanho, bloco);           // Copia um bloco para memória
        j = criarBufferMatriz(BlocosLength);                    // Cria uma matriz
        j = frequenciaCalculo(v, tamanho, bloco, j);                    // Guarda em memória as frequencias de cada simbolo para cada bloco
        caracterRLE = compressaoRLE(file_name, v, tamanho, bloco); // Comprime e escreve no ficheiro .rle bloco a bloco && retorna o numero de caracteres final.

        taxa = taxaCompressao(caracterRLE, v); // Calcula a taxa de compressão
        blocosRLE[RleB] = caracterRLE;         // Guarda o tamanho do bloco depois de comprimido
    }

    char* nomeFile = malloc(sizeof(char)*50);
    strcpy(nomeFile, nomeRLE(file_name));

    funcaoFrequenciaEscrita(file_name, j, 'R', FileLength, size, BlocosLength);       // Função que cria e cálcula as frequencias de cada simboolo em cada bloco
    funcaoFrequenciaEscritaRLE(nomeFile, 'R', BlocosLength, blocosRLE);
    imprimeTerminal(file_name, 'R', BlocosLength, FileLength, size, taxa, blocosRLE); // Função para escrever texto informativo no terminal
```

Figura 6: Código forca_compressao

Esta função é semelhante à anterior, diferenciando-se na ausência das restrições que impediam a função de executar a compressão RLE consequente de uma taxa de compressão inferior a 5%.

Quanto às limitações do código gerado para este módulo, penso que existem demasiados acessos aos ficheiros. Por exemplo, após a execução da compressão .rle, em vez de guardar o resultado logo no ficheiro de saida poderiamos guardar este num buffer de saida que permitiria calcular as suas frequências em memória em detrimento de voltar a ler tudo da memória. Além disso, foram utilizadas funções que poderiam ser trocadas por outras semelhantes mas mais optimizadas o que permitiria evitar possíveis erros num fututo. Percebo também que há em alguns momentos alocação de memória desnecessária mas que por falta de conhecimento/tempo não me permitiu retificar.

3.2 Módulo B

Neste módulo, inicialmente a informação é lida do ficheiro para uma matriz inteiros. Esta é composta de arrays com capacidade para 256 inteiros. Os indíces na matriz representam os vários blocos e os indices dos arrays os códigos ASCII e o valor do inteiro a frequência desse mesmo símbolo. Esta informação é então transformada numa lista ligada onde cada nodo tem a informação relativa ao símbolo e a sua respectiva frequência para que operações como organizar por ordem decrescente e por ordem lexicográfica se torne possível. Após isto tudo que se tem que fazer é aplicar o algoritmo do shannon-fannon repetidamente para todos os blocos e criar progressivamente as strings binárias (na forma de lista ligada). Quanto à parte crítica, acredito que a função de descoberta do tamanho do último bloco é pouco eficiente.

3.3 Módulo C

A nossa abordagem ao módulo C regeu-se em torno da utilização de arrays e matrizes para um mais fácil acesso aos dados presentes nos ficheiros.

Para o tratamento do ficheiro .cod utilizamos uma função que passou o conteúdo do mesmo para uma string, para ser mais fácil o acesso ao seu conteúdo. Após esse tratamento inicial, a string irá ser utilizada para carregar os códigos dos caracteres presentes para uma matriz ordenada por blocos. Esta matriz está ordenada por linhas e cada uma destas linhas tem 256 possíveis elementos cada um correspondendo ao código do carácter cujo valor inteiro é o índice do elemento.

Para o tratamento do ficheiro original criamos uma função que coloca todo o conteúdo do ficheiro na memória, dividindo o ficheiro em buffers bloco-a-bloco.

Em relação à impressão do ficheiro final .shaf criamos uma função que passava um buffer de caracteres vindo do ficheiro original para um array com a sequência de 0's e 1's correspondentes aos códigos de cada um dos caracteres do buffer. Posteriormente, utilizamos uma função que, bloco-a-bloco, chama a função anteriormente mencionada e pega em 8 elementos do array devolvido e imprime um carácter em ASCII que corresponde a essa mesma sequência.

O nosso código é limitado em relação a ficheiros de tamanho mais elevado devido à utilização excessiva de memória aquando da leitura dos ficheiros. Também há, possívelmente, lapsos da nossa parte no que toca à otimização do código.

3.4 Módulo D

A abordagem ao módulo D consistiu em, receber os ficheiros, colocando os mesmos em memória através do buffer. Inicialmente, com o ficheiro cod, este buffer com o conteúdo do ficheiro seria dividido, criando matriz em que cada linha corresponderia a um bloco do ficheiro e cada coluna a um elemento desse mesmo bloco. De seguida, cada bloco seria utilizado para construir listas de códigos associadas a um caracter que posteriormente originariam uma espécie de árvore de procura em que, dado um código, seguiria 1 para a direita e 0 para a esquerda, terminando com o caracter a colocar no caso de chegar aquele nodo.

Relativamente ao ficheiro shaf, funcionaria da mesma forma para trabalharmos com o mesmo em memória. Neste caso, devido à leitura ter de ser realizada em binário, em cada bloco traduzimos os caracteres para um array com os caracteres em binário. De seguida, relacionamos a leitura efetuada do ficheiro shaf com uma matriz com as árvores criadas através do ficheiro cod.

O ficheiro rle teria uma descodificação semelhante com a especificidade de que a repetição de caracteres se encontra comprimida, pelo que definimos funções que descodificassem este ficheiro.

4 Tabela de resultados

4.1 Módulo A

Nome Ficheiro	Tamanho Blocos	Force Compressão	Taxa Compressão	Tempo Execução
aaa.txt	2048 Bytes	0	51%	0.339 ms
aaa.txt.rle	2048 Bytes	1	51%	0.347 ms
Shakespeare.txt	640 Bytes	0	2%	30.698 ms
Shakespeare.txt.rle	640 Bytes	1	2%	105.729 ms
bbb.zip	8 MBytes	0	???	15295.828 ms
bbb.zip	8 MBytes	1	???	14633.421 ms

4.2 Módulo B

Ficheiros testados	Número de blocos	Tamanho Bloco	Tempo de execução(milissegundos)
aaa.txt.freq	2	2048	3.001
aaa.txt.rle.freq	2	2048	0.229
Shakespeare.txt.freq	9	655360	4.484
Shakespeare.txt.rle.freq	9	638735	10.465
bbb.zip.freq	90	8388608	179.523
bbb.zip.rle.freq	90	8418559	145.285

4.3 Módulo C

Ficheiros testados	Número de blocos	Compreensão global(%)	Tempo de execução(milissegundos)
aaa.txt	2	77.66	0.711000
aaa.txt.rle	2	66.09	0.887000
Shakespeare.txt	9	39.38	703.028015
Shakespeare.txt.rle	9	37.80	706.390991

4.4 Módulo D

Ficheiro de entrada	Ficheiro de saída	Tipo de descompressão	Número de blocos	Tempo de execução (milis
aaa.txt	aaa.txt	SF	2	0.195
aaa.txt.rle	aaa.txt.rle	SF	2	0.255
aaa.txt.rle	aaa.txt	RLE	2	0.364
aaa.txt.rle	aaa.txt	SF + RLE	2	0.261

5 Conclusão

O trabalho final apresentado encontra-se funcional e, enquanto grupo, consideramos que trabalhamos de forma a alcançar os principais pontos do trabalho.

5.1 Módulo A

O Módulo A cumpre os seus objetivos na maioria dos casos pretendidos, apresentando apenas alguns erros em certos ficheiros. As dificuldades neste trabalho consistiram também na desinformação acerca do manuseamento de ficheiros e binário.
Gostaria de ter conseguido mudar a maneira como estruturei o código inicialmente de forma a optimizar e evitar possíveis falhas ao executar este módulo com ficheiros mais complexos.

5.2 Módulo B

Em suma, acreditamos que a maioria do código escrito é eficiente e funciona na totalidade, no entanto, podia-se beneficiar de uma melhor estrutura de forma a ficar mais fácil de se compreender.

5.3 Módulo C

O Módulo C executa a maioria dos exemplos disponibilizados pelos professores apesar de não conseguirmos executar bem o ficheiro bbb.zip. Então, foi um trabalho de um calibre exigente e achamos que procedemos da melhor forma que conseguimos ver.

5.4 Módulo D

Relativamente ao módulo D consideramos que, apesar do código não ser executável para todos os ficheiros fornecidos pelos docentes, o trabalho desenvolveu quer o raciocínio quer a facilidade com a linguagem C. No entanto, consideramos que com algum tempo extra poderíamos ter resolvido os problemas de acessos à memória.

6 Bibliografia

Os principais recursos utilizados foram os seguintes:

- Tabela ASCII
- $h$ttps://www.tutorialspoint.com/index.htm
- $h$ttps://pt.stackoverflow.com/