Universidade do Minho
Escola de Engenharia

SHAFA

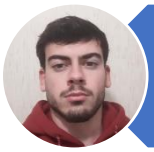
Universidade do Minho
Mestrado Integrado em Engenharia Informática
Comunicação de Dados
Grupo 10
3 de janeiro de 2021


Carlos Dias A93185


João Vieira A93170


Francisco Novo A89567


Joaquim Roque A93310


Francisco Izquierdo A93241


José Magalhães A93273


Duarte Lucas A89526


Tiago Ribeiro A93203

Índice

Organização do Trabalho

Para este trabalho decidimos não optar pela sugestão dos docentes, que recomendavam a divisão dos vários módulos do trabalho pelos elementos do grupo. Sendo assim, achamos que seria mais eficiente todos os elementos do grupo tomarem parte da realização de cada módulo. No final do projeto definimos a divisão do grupo em pares para a defesa do trabalho.

A organização dos elementos do grupo para a defesa é a seguinte:

- **Módulo f** → Francisco Novo (A89567) e Joaquim Roque (A93310);
- **Módulo t** → Duarte Lucas (A89526) e Tiago Ribeiro (A93203);
- **Módulo c** → Francisco Izquierdo (A93241) e João Vieira (A93170);
- **Módulo d** → Carlos Dias (A93185) e José Magalhães (A93273).

Discussão, estratégias e otimizações

No módulo f o nosso grupo optou por gerar o ficheiro do tipo freq a partir do ficheiro original. No módulo t e c usamos a estrutura Caracter (Figura 1) que é uma lista ligada, onde podem ser guardados bits.

```
typedef struct nodo{
    unsigned char value;
    struct nodo *prox;
}*Caracter;
```

Figura 1- Estrutura Caracter

No módulo d, inicialmente, procurávamos uma sub-string do bloco de bits no array com as codificações de cada caracter e se não fosse encontrada o tamanho dessa sub-string era incrementado. Chegamos à conclusão que procurar desta maneira era extremamente ineficiente, em certos ficheiros a descompressão demorava várias horas. De seguida, percorremos o array e procuramos todas as codificações dos caracteres no local em que nos encontrávamos no bloco, isto melhorou 7,5 vezes o tempo de execução do código, mas ainda não era satisfatório. Por fim, decidimos utilizar árvores binárias (Figura 2), para melhorar o desempenho da descompressão, guardando nelas os "códigos SF" de cada caracter.

```
typedef struct TreeBinary{
    unsigned char letra;
    struct TreeBinary *esq, *dir;
}*ABin;
```

Figura 2- Estrutura Abin

Funções principais

Durante o trabalho deparamo-nos com funções que utilizamos em vários módulos, como a give_Number que é usada para retirar números dos vários tipos de ficheiro e a digits_in_ASCII que passa um número para uma lista com todos os dígitos desse número.

No módulo f as principais funções são: verificaCompressao, buscaBloco, tamanhoDoBlocoComprimido, compressaoRLE e freq_Original. Inicialmente, a função verificaCompressao calcula a taxa de compressão do primeiro bloco do ficheiro original. De seguida, a função buscaBloco tem como objetivo procurar um bloco no ficheiro original e a função tamanhoDoBlocoComprimido calcula o tamanho de um bloco depois de realizada a compressão. Com estas funções podemos efetuar a compressão seguindo o formato RLE, num bloco com a função compressaoRLE. Por fim, a função freq_Original gera o ficheiro das frequências (.freq) do ficheiro original.

No módulo t revelaram grande importância as seguintes funções: gera_Pares, reverse_Sort, recursive_SF, find_Meio e add_bit_to_code. A função gera_Pares retorna um array que nos índices pares contém os números dos caracteres da tabela ASCII e nos índices ímpares as frequências com que ocorrem, sendo que a função reverse_Sort ordena os pares desse array por ordem decrescente das frequências dos caracteres. As funções recursive_SF, find_Meio e add_bit_to_code apresentam um papel fundamental na criação do código SF, sendo que a função find_Meio divide um conjunto pertencente ao array de pares em dois subconjuntos, na qual a soma das frequências de cada subconjunto seja sempre o mais próximo possível da metade das frequências do conjunto. A função add_bit_to_code atribui um bit para o primeiro subconjunto e outro para o segundo subconjunto, sendo que este método é aplicado recursivamente na função recursive_SF.

No módulo c as funções relevantes são: preenche_Array_Cod_C, preenche_bloco_shaf e divide_bloco. A função preenche_Array_Cod_C constrói um array de estruturas Caracter e coloca nelas os códigos SF de cada caracter. A função preenche_bloco_shaf vai preenchendo o bloco

correspondente do ficheiro do tipo shaf. A função divide_bloco divide o bloco em conjuntos de 8 bits e transforma cada um no seu respetivo byte, retornando assim o array com bytes calculados.

No módulo d resumem-se as seguintes funções essenciais: preenche_ABin, translate_2_bits, descodificaSF_Super_Otimizada, calcula_tam_bloco_original, descomprime_RLE. A função preenche_ABin preenche uma árvore binária onde serão guardados implicitamente os códigos SF de cada caracter. A função translate_2_bits passa os bytes de um bloco para a sequência binária correspondente, sendo que depois será possível proceder à descodificação do código SF que será realizada pela função descodificaSF_Super_Otimizada que percorre um bloco de bits e usa cada bit para navegar na árvore binária criada anteriormente, onde irá encontrar um caracter. Se receber o bit 0 continuará a procura na árvore da esquerda e se receber o bit 1 continua na da direita e percorre-a até a árvore para onde for procurar estiver a NULL, aí encontrará o caracter correspondente ao código SF. A função calcula_tam_bloco_original irá calcular o tamanho do ficheiro original sem compressão RLE, sendo depois capaz de avançar para a descompressão do ficheiro que será realizada pela função descomprime_RLE que faz o processo inverso à compressão RLE, bloco a bloco, obtendo assim cada bloco original.

Resultados

Na seguinte tabela constam os tempos de execução (em milésimos de segundo) de vários ficheiros nos quatro módulos:

		aaa.txt (pequeno)	Shakespeare.txt (médio)	bbb.zip (grande)
Módulo f (.rle, .rle.freq e .freq)	.rle , .rle.freq	32	406	28597
	freq		70	6922
Módulo t (.cod e .rle.cod)	.cod		24	264
	.rle.cod	19	134	189
Módulo c (.shaf e .rle.shaf)	.shaf		1432	203359
	.rle.shaf	16	2264	257867
Módulo d (ficheiro original ou .rle)	Ficheiro original (.shaf)		569	150732
	Ficheiro original (.rle.shaf)	6	843	159772
	.rle	8	778	147600

Conclusões

Por fim, neste trabalho fizemos algumas otimizações, contudo podíamos ter feito ainda mais para melhorar os tempos de execução. Além disso, gostaríamos de ter implementado a otimização por execução paralela e a codificação binária SF com matrizes de bytes. Contudo, o pouco tempo foi prejudicial para alcançarmos um trabalho ainda melhor.

Bibliografia

- Função fsize disponibilizada pelos docentes;