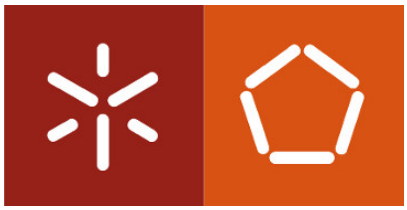


Universidade do Minho

Escola de Engenharia

Mestrado Integrado em Engenharia de Telecomunicações e Informática

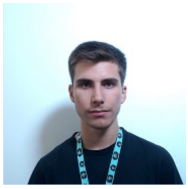
Serviços de Rede e Aplicações Multimédia

Trabalho Prático Nº1 – Compressão LZWd

Professor Dr. Bruno Alexandre Fernandes Dias

26 de junho de 2023

Grupo:



Rui Cunha - A93079
a93079@alunos.uminho.pt

Francisco Martins - A93079
a93079@alunos.uminho.pt

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Introdução

Este relatório está inserido no âmbito da Unidade Curricular Serviços de Rede e Aplicações Multimédia, do 2ºsemestre do 4ºano do Mestrado Integrado em Engenharia de Telecomunicações e Informática, como resposta ao problema apresentado pelo professore. Para realizar este trabalho recorre-se as bases dos trabalho prático inicial que visa promover a familiarização dos alunos com as ferramentas utilizadas neste trabalho e ao material existente na *BlackBoard* [1].

O objetivo deste trabalho consiste em implementar uma aplicação codificadora de ficheiros que implemente o algoritmo LZWd e que calcule alguns dados estatísticos sobre o processo de codificação utilizando a linguagem C. O projeto é constituído pelo seguinte source e header presentes na imagem que se segue. A pasta ”files”contém os ficheiros que serão comprimidos.

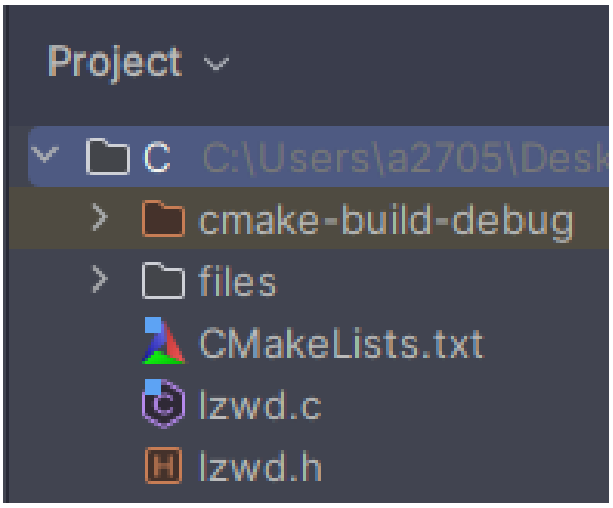


Figura 1: Source e Header do projeto.

2 Estratégias utilizadas

Esta secção do relatório tem como objetivo apresentar as principais estratégias do código desenvolvido em linguagem C. O código em questão é uma poderosa ferramenta que visa facilitar e otimizar diversas tarefas, proporcionando eficiência e precisão no processamento de dados. Foi utilizado o algoritmo fornecido pelo professor na construção do código. O grupo decidiu utilizar a struct, apresentada na imagem em baixo, para usar como estrutura do dicionário, em que o apontador ”value”armazena todos os bytes do padrão e a variável ”size”indica o tamanho do padrão em bytes.

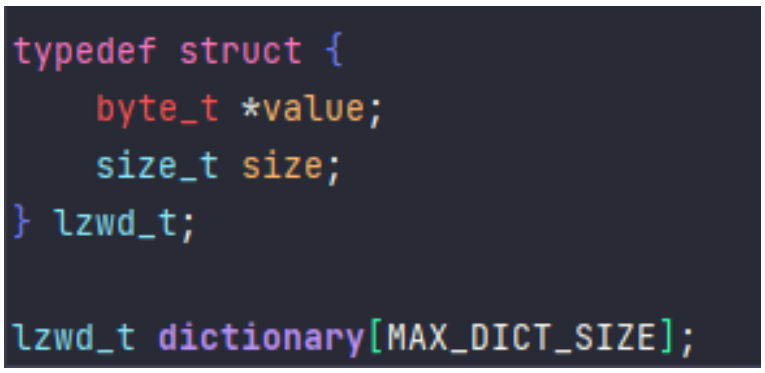


Figura 2: `Struct lzwd_t` .

O código começa com a abertura dos ficheiros de entrada e saída, e inicia o dicionário com 256 padrões de símbolos individuais. De seguida, o ficheiro de entrada é lido em blocos de dados de 64KByte que irão ser processados individualmente e são criadas duas variáveis lzwd_t, **Pa** e **Pb**, usadas para armazenar os símbolos em cada processamento no bloco de dados. Depois da inserção dos símbolos nas variáveis, **Pa** e **Pb**, é feita a busca pelo maior padrão conhecido depois de **Pa**. Caso não seja encontrado nenhum, **Pb** continua com o mesmo padrão.

De seguida, é procurado o código correspondente ao símbolo **Pa** para ser enviado para o output e, caso o dicionário não esteja cheio, é acrescentado ao dicionário, em cada iteração, um ou mais novos padrões formados a custa dos dois padrões já conhecidos, **Pa** e **Pb**. Caso contrario, este será reiniciado com os 256 padrões iniciais e o bloco de dados continuará a ser processado.

3 Funções implementadas

Nesta secção do relatório, vamos explorar as funções em C, discutindo sua definição, declaração, chamada e retorno de valores. As funções são elementos cruciais na programação, pois nos permitem organizar o código em blocos reutilizáveis, promovendo a modularidade e facilitando a manutenção do programa.

Na figura abaixo, podemos ver a função utilizada para criar o dicionário com os 256 padrões iniciais. Por cada valor introduzido na variável ”value”, é primeiramente feita uma alocação de memoria. Caso a alocação de memoria não seja bem sucedida, é retornado ”false”que indica ao programa que houve um erro e fecha.

```
// Function used to create a default dictionary
bool create_dictionary(lzwd_t dict[]) {
    dict_size = 256;

    for (int i = 0; i < dict_size; i++) {        // From 0 to 255
        dict[i].value = (byte_t *) malloc( Size: sizeof(byte_t));
        if (dict[i].value == NULL) {
            return false;
        }
        dict[i].value[0] = i;      // Position 0 will be its own index
        dict[i].size = 1;
    }

    // All remaining indexes will be 0
    for (size_t i = dict_size; i < MAX_DICT_SIZE; i++) {      // From 256 to 65536
        dict[i].value = NULL;
        dict[i].size = 0;
    }

    return true;
}
```

Figura 3: Criação do dicionário.

Na figura 4, podemos observar a função que efetua a pesquisa pelo dicionário pelo maior padrão conhecido depois de **Pa**. O ciclo *while* percorre o bloco de dados até que o valor *counter* (a posição do bloco de dados) mais 1 seja menor que o tamanho do bloco de dados.

```
while (counter + j < block_size) {
    if (find_dictionary_sequence( map: &Pb,  pattern: block[counter + j],  size_of_pattern: Pb.size + 1) != -1) {
        Pb.value[Pb.size] = block[counter + j];
        Pb.size = Pb.size + 1;

        j++;
    } else {
        break;
    }
}
```

Figura 4: Função para procurar o maior padrão conhecido.

Em cada iteração, é verificado se existe o padrão **Pb** com o *block[counter + j]* utilizando a função ”find_dictionary_sequence”apresentada na figura 5. Esta função percorre o dicionário e executa a função ”equal_bytes”para verificar que os padrões coincidem. Caso o padrão exista, o byte do *block[counter + j]* é adicionado ao **Pb** e o size é incrementado. Se o padrão não existir, terminasse o ciclo *while*.

```
// Function used to find the index of the pattern
int find_dictionary_sequence(lzwd_t *map, byte_t pattern, size_t size_of_pattern) {
    map->value[size_of_pattern - 1] = pattern;
    map->size = size_of_pattern;

    for (int i = 0; i < dict_size; i++) {
        if (dictionary[i].size == size_of_pattern) {
            if (equal_bytes(map,  index_i: i)) {
                return i;
            }
        }
    }

    return -1;

}
```

Figura 5: Função para verificar se padrão existe no dicionário.

Na figura 6, é apresentada a função usada para acrescentar no dicionário um ou mais novos padrões formados a custa dos dois padrões **Pa** e **Pb**. O ciclo *while* percorre por todos os bytes do padrão **Pb** e verifica se o padrão **Pa** mais o byte existem no dicionário utilizando a função "search_dictionary_pattern"que podemos observar na figura 7. Caso o padrão não exista, é alocada memoria para pudermos introduzir o padrão no dicionário e o *size* irá ser igual a soma do *size* dos dois padrões. Caso o padrão exista no dicionário, o index é incrementado para passar para o próximo *byte* do **Pb**. No inicio do ciclo, é feita a verificação do tamanho do dicionário, caso este tenha atingido a capacidade máxima é feita a limpeza do dicionário e são criados os 256 padrões iniciais.

```
int index = 1;

while (index <= j) {
    // Add the pattern in dictionary
    if (dict_size == MAX_DICT_SIZE) {

        dict_reset++;

        free_map( map: dictionary);
        // Creates the default dictionary
        if (!create_dictionary( dict: dictionary)) {
            return 0;
        }

    }
    Pb.size = index;
    if ((search_dictionary_pattern(Pa, Pb) == -1)) {

        dictionary[dict_size].value = (byte_t *) malloc( Size: sizeof(byte_t));
        if (dictionary[dict_size].value == NULL) {
            printf( format: "\nError! memory not allocated.");
            exit( Code: 0);
        }
        dictionary[dict_size].size = Pa.size + Pb.size;

        int i = 0;

        for (int i_1 = 0; i_1 < Pa.size; i_1++) {
            dictionary[dict_size].value[i] = Pa.value[i_1];
            i++;
        }
        for (int i_2 = 0; i_2 < Pb.size; i_2++) {
            dictionary[dict_size].value[i] = Pb.value[i_2];
            i++;
        }
        dict_size++;
    }
    index++;
}
```

Figura 6: Função usada para adicionar um ou mais padrões com **Pa** e **Pb**.

Na função da figura 7, é criado uma variável temporária *temp_pattern* que é o padrão **Pa** mais o padrão **Pb** para ser feita a pesquisa do padrão (**Pa** + **Pb**) pelo dicionário. Se o padrão existir é retornado o posição do padrão no dicionário.

```
// Search in the dictionary if the pattern Pa + Pb exists
int search_dictionary_pattern(lzwd_t Pa, lzwd_t Pb) {
    lzwd_t temp_pattern;
    temp_pattern.size = Pa.size + Pb.size;

    temp_pattern.value = (byte_t*) malloc( Size: sizeof (byte_t) * (temp_pattern.size));
    if(temp_pattern.value == NULL) {
        printf( format: "\nError! memory not allocated.");
        exit( Code: 0);
    }

    int i = 0;

    for (int i_1 = 0; i_1 < Pa.size; i_1++) {
        temp_pattern.value[i] = Pa.value[i_1];
        i++;
    }
    for (int i_2 = 0; i_2 < Pb.size; i_2++) {
        temp_pattern.value[i] = Pb.value[i_2];
        i++;
    }

    for (int x = 0; x < dict_size; x++) {
        for (int j = 0; j < temp_pattern.size; j++) {
            if (equal_bytes( map: &temp_pattern,  index_i: x)) {
                free( Memory: temp_pattern.value);
                return x;
            }
        }
    }

    return -1;
}
```

Figura 7: Função usada para verificar se o padrão (**Pa** + **Pb**) existe.

4 Testes

De modo a testar o código, é usado como ficheiro de entrada o ficheiro ”exemplo.txt”, que é um ficheiro escolhido pelo grupo com 17023 bytes, para a verificação da compressão. Como podemos verificar o ficheiro demorou 1.79 segundos a concluir a compressão e o tamanho final do dicionário é de 14228 padrões.

```
Compression successful!
----------------------------------------
Size of block: 17023
----------------------------------------
Number of blocks: 1
----------------------------------------
Dictionary resets: 0
----------------------------------------
Size of dictionary: 14228
----------------------------------------
Number of outputs: 14013
----------------------------------------
Time elapsed: 1.79 seconds
----------------------------------------
```

Figura 8: Compressão efetuada a ficheiro de 17023 bytes.

Na figura 9, podemos verificar que os códigos dos padrões são guardados como uma sequencia de dígitos decimais, separados por linhas.

```
13,10,84,105,116,108,101,58,32,84,104,101,32,67,111,109,112,261,116,267,87,111,114,107,115,32,111,
101,97,114,101,256,256,65,117,116,104,279,263,287,289,291,293,295,297,299,301,303,306,256,76,97,11
275,114,32,115,101,116,32,101,110,99,111,100,105,343,263,85,84,70,45,56,256,45,45,393,394,396,399,
332,337,308,256,98,121,286,288,290,292,294,296,298,300,302,304,469,308,32,32,503,503,269,110,275,5
504,65,76,76,226,128,153,83,286,69,565,264,72,65,84,350,78,68,571,574,559,504,534,536,84,82,65,71,
579,615,608,65,570,89,79,85,32,76,73,75,536,73,84,608,612,67,79,77,618,620,621,350,82,82,79,82,548
573,73,541,698,741,748,578,77,636,84,44,32,80,752,78,67,536,708,68,69,78,77,65,82,75,698,70,73,714
72,698,83,69,700,583,829,844,882,885,887,890,893,898,902,926,670,70,838,946,952,898,73,70,534,869,
1088,534,871,633,877,1111,1120,1084,1004,612,69,73,71,72,1032,946,74,79,72,78,813,748,74,85,670,75
66,901,1365,669,79,83,694,739,745,708,843,67,782,1032,702,663,85,82,536,899,82,32,1445,1447,69,698
1361,570,708,1530,583,83,713,662,1454,73,584,85,77,1486,32,1497,1218,84,1377,68,1448,65,77,608,77,
```

Figura 9: Conteúdo do ficheiro comprimido output.txt.lzwd.

De seguida, decidimos testar com um ficheiro de tamanho mais elevado, 84124 bytes. Como podemos observar a compressão demorou 44.34 segundos e o tamanho final do dicionário é 62312 padrões.

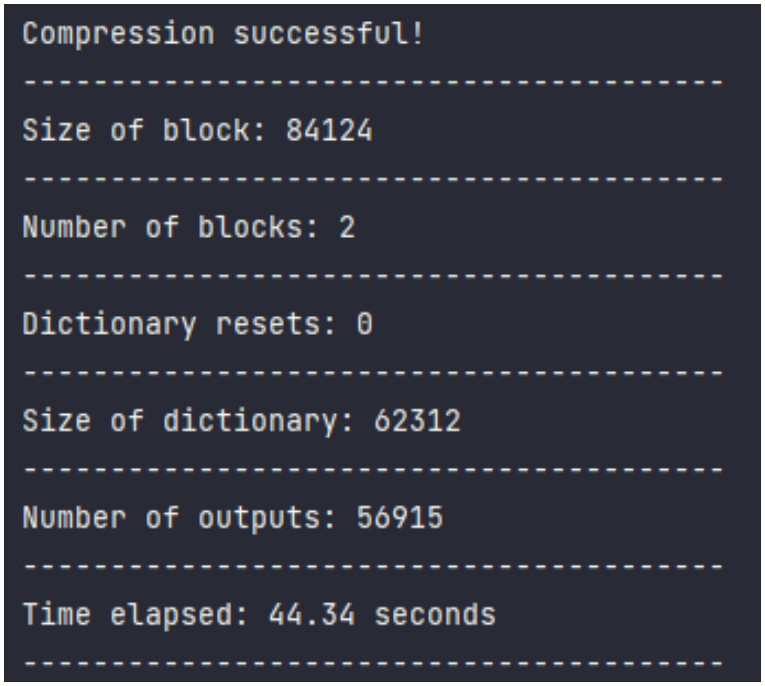


Figura 10: Compressão efetuada a ficheiro de 84124bytes.

Na terceira tentativa, o grupo decidiu testar usando um ficheiro de 211269 bytes, e como podemos observar na figura 11 o processo demorou 153.17 segundos a terminar, ocorrendo 2 reinícios de dicionário sendo que o dicionário final terminou com 19654 padrões.

```
Compression successful!
----------------------------------------
Size of block: 211269
----------------------------------------
Number of blocks: 4
----------------------------------------
Dictionary resets: 2
----------------------------------------
Size of dictionary: 19654
----------------------------------------
Number of outputs: 157377
----------------------------------------
Time elapsed: 153.17 seconds
----------------------------------------
```

Figura 11: Compressão efetuada a ficheiro de 211269 bytes.

Finalmente, o grupo decidiu testar usando um ficheiro disponibilizado pelo professor ”example.txt”, com quase 5 Mbytes e, observando pela figura 12, podemos ver que o processo terminou ao fim de 4087.51 segundos, ocorrendo 32 reinícios de dicionário sendo que o dicionário final terminou com 54255 padrões.

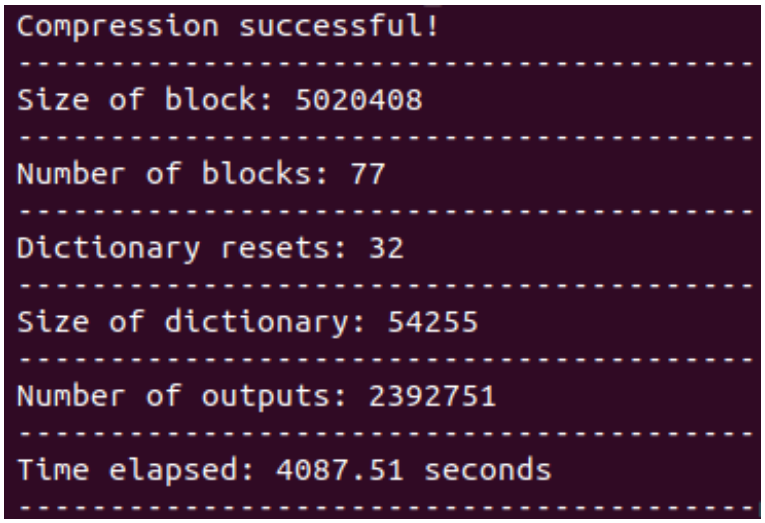


Figura 12: Compressão efetuada a ficheiro de 5020408 bytes.

5 Conclusão

A concretização deste projeto foi bastante útil para aprofundar e aplicar os conhecimentos adquiridos nas aulas ao longo do semestre.
Numa primeira fase, obteve-se dificuldades em entender como trabalhar com alocação da memória e quais as suas funcionalidades, fazendo com que não fosse possível atingir alguns objetivos.

Deste modo, é colossal o conhecimento que se levou após a realização deste projeto.

Referências

[1] BlackBoard.