



Universidade do Minho
Escola de Engenharia

MESTRADO EM ENGENHARIA DE TELECOMUNICAÇÕES E INFORMÁTICA

SERVIÇOS DE REDE E APLICAÇÕES MULTIMÉDIA

TRABALHO PRÁTICO Nº1 – COMPRESSÃO LZWD



Camila Pinto - PG53712

Barbara Fonseca - PG53677

Eduarda Dinis - PG53793

Conteúdo

1	Introdução	2
2	Estratégias Adotadas	3
2.1	Estrutura do dicionário	3
2.2	Tipo de dicionário	4
2.3	Tipo de limpeza do dicionário	5
2.4	Função de compressão	5
3	Análise Crítica	7
4	Análise de testes	10
4.1	Ficheiros de texto	10
4.2	PDF	11
4.3	Imagem	12
5	Manual de Utilização	12
6	Conclusão	13

1 Introdução

O presente relatório foi desenvolvido no âmbito do projeto da Unidade Curricular de Serviços de rede e aplicações multimédia, do 2º semestre do 1º ano do Mestrado em Engenharia de Telecomunicações e Informática. O principal objetivo deste trabalho, é implementar uma aplicação codificadora de ficheiros utilizando o algoritmo LZWdR, e calcular alguns dados estatísticos sobre o processo de codificação.

Num contexto mais amplo, a compressão consiste em converter um conjunto de dados num código para poupar a necessidade de armazenamento e transmissão de dados, tornando mais fácil a transmissão dos mesmos. Este processo implica a redução do tamanho de um ficheiro de grandes dimensões para um tamanho menor, facilitando a sua transmissão (1). Dentro deste contexto, existem diversos tipos de algoritmos de codificação, sendo que neste projeto será estudada a codificação por padrões.

A codificação por padrões é usada para identificar padrões repetitivos nos dados e substituí-los por códigos mais curtos, resultando numa representação mais compacta dos dados. Estes códigos são posteriormente utilizados para reconstruir os dados originais durante a descompressão (2). Esta também é caracterizada pelo uso de um dicionário.

A codificação por padrões está dividida em vários algoritmos, destacando-se o algoritmo LZ78, do qual a codificação LZW é uma variante. A estratégia da codificação LZW consiste em gerar códigos simples, conhecidos como índices, que representam os padrões encontrados (3).

A partir desta variante, surgiu a codificação LZWdR, desenvolvida pela Universidade do Minho. Esta difere da LZW na forma como procura padrões, uma vez que a LZWdR adiciona um ou mais padrões utilizando o conhecimento de dois padrões(devido à inversão), enquanto o LZW acrescenta apenas um novo padrão em cada iteração, formado por um padrão já conhecido e o símbolo seguinte. Este método resulta num crescimento mais rápido do dicionário do LZWdR em comparação com o dicionário do LZW, o que potencialmente pode aumentar o desempenho, embora exija mais memória durante a execução (3).

2 Estratégias Adotadas

2.1 Estrutura do dicionário

Para a implementação do algoritmo de codificação LZWdR, é necessário desenvolver uma estrutura denominada dicionário, responsável por armazenar os padrões de dados encontrados durante o processo de busca juntamente com o seu código, tornando a representação dos dados mais compacta e proporcionando uma maior eficiência na compressão.

Existem diferentes opções para implementar o dicionário, incluindo algoritmos de ordenação clássicos (como as tries), tabelas de hash e arrays de índices. As tabelas de hash são mais eficientes em termos de tempo, fornecendo operações de inserção, pesquisa e remoção em tempo constante, o que resulta numa busca por padrões rápida e eficiente. Além disso, são mais simples de implementar e não exigem a manutenção da ordem dos padrões. Embora possam sofrer de colisões, geralmente são mais eficientes em termos de espaço do que as tries (4).

Por outro lado, as tries podem consumir mais memória, especialmente em grandes conjuntos de dados. A implementação de uma trie pode ser mais complexa do que uma tabela de hash, pois requer uma estrutura de árvore e operações de inserção, pesquisa e exclusão mais complexas. Além disso, resolver colisões numa trie pode afetar o desempenho (4).

Desta forma, para implementar o dicionário da codificação optámos por uma Hash Table, como é demonstrado na figura seguinte:

```
typedef struct Ht_item
{
    unsigned int key; // código atribuído a uma sequência de entrada
    std::string value; // padrão
    Ht_item *next; // apontador para o próximo item na lista ligada
} Ht_item;

/// @brief Estrutura que representa a hashtable
typedef struct HashTable
{
    Ht_item **items; // array de apontadores a apontar para Ht_item
    unsigned int count; // Contador de itens na tabela
} HashTable;
```

Figura 1: Estrutura do dicionário.

Para lidar com o desafio das colisões, optámos por utilizar o método de encadeamento separado. Neste método, múltiplos elementos com chaves distintas são armazenados na mesma posição da tabela, por meio de uma lista ligada. Na nossa Hash Table, definimos um array de *pointers* para a estrutura Ht_item, cada elemento deste array representa um slot na tabela, permitindo-nos organizar os elementos de acordo com as suas chaves.

A estrutura `Ht_item` representa um elemento e contém três campos:

- `key`: um código atribuído a uma sequência de entrada.
- `value`: uma string que representa o valor associado à chave.
- `next`: um apontador para o próximo item na lista ligada.

O `next` é usado para implementar o encadeamento separado, quando ocorre uma colisão, isto é, quando dois elementos têm o mesmo índice na tabela, este campo permite-nos encadear os elementos, criando uma lista ligada que nos permite armazenar múltiplos elementos na mesma posição da tabela.

2.2 Tipo de dicionário

Para inicializar o tipo de dicionário, implementámos a estratégia de o inicializar com os 256 padrões iniciais (ASCII). Na figura seguinte, isto é demonstrado pela função `create_table`, que cria uma table com as posições vazias e posteriormente preenche as 256 posições iniciais.

```
// Função para inicializar tabela hash, new é usado para criar memória dinamicamente
HashTable *create_table(unsigned long max_dic_size, int init_entrys)
{
    HashTable *table = new HashTable; // alocar memória para a tabela
    table->items = new Ht_item *[max_dic_size]; // alocar um novo apontador
    table->count = 1; // inicializar contador

    // Inicializa todos os slots com nullptr, ou seja, contador nulo
    for (unsigned long i = 0; i < max_dic_size; i++)
    {
        table->items[i] = nullptr;
    }

    // Preenche com os 256 padrões iniciais
    for (int f = 0; f < init_entrys; f++)
    {
        Ht_item *item = new Ht_item; // Aloca memória para o item
        item->key = f; // Define a chave como índice
        item->value = std::string(1, static_cast<char>(f)); // Converte o índice para string e define como valor
        table->items[f] = item; // Adiciona o item à tabela
        table->count++; // Incrementa o contador
    }

    return table;
}
```

Figura 2: Função para inicializar o dicionário.

2.3 Tipo de limpeza do dicionário

Para implementar a valorização da limpeza do dicionário, seguimos a estratégia de fazer reset, ou seja, a limpeza completa do dicionário, pelo que quando o tamanho máximo do dicionário é atingido, este volta ao dicionário inicial, com apenas os 256 padrões. Desta forma, na função *insert* verificamos se o tamanho do dicionário é atingido e caso seja, a função *reset* é chamada. A função *reset* é apresentada na figura seguinte.

```
// Função para resetar ("limpar") a hash table mantendo os 256 primeiros itens
int reset(HashTable *hashtable, unsigned long max_dict_size)
{
    for (unsigned long i = 256; i < max_dict_size; ++i)
    {
        Ht_item *current = hashtable->items[i];
        while (current != nullptr)
        {
            Ht_item *temp = current;
            current = current->next;
            delete temp; // Liberta a memória alocada para o item
        }
        hashtable->items[i] = nullptr; // Marca como nullptr para indicar que está vazio
    }
    //contagem de itens para 256
    hashtable->count = 256;
    num_dict_resets++;
    return num_dict_resets;
}
```

Figura 3: Função para limpar o dicionário.

A função *reset* é responsável por reiniciar completamente o dicionário quando o tamanho máximo é alcançado. Percorre os slots da tabela, libertando a memória alocada para os itens existentes e marcando os slots como vazios. Em seguida, atualiza a contagem de itens para o valor inicial e incrementa o contador de resets do dicionário, utilizado posteriormente para a estatística de número de vezes que o tamanho máximo do dicionário foi atingido.

2.4 Função de compressão

Para implementar a função de compressão, seguimos o algoritmo descrito no enunciado, conhecido como LZWdR (algoritmo não otimizado). Inicialmente, o algoritmo define o primeiro padrão, designado por "Pa", como sendo o primeiro carácter da sequência, enquanto o índice (*ind*) acompanha a posição atual de leitura.

Em seguida, dentro do ciclo, é calculado o código para "Pa" utilizando a função *search*, para procura os padrões mais longos já existentes no dicionário. Se esses padrões já existirem, o valor de "Pb" é atualizado e a variável de padrões encontrados é incrementada, sendo posteriormente utilizada para estatísticas.

Após esgotar a busca por padrões, o código de "Pa" é enviado para a saída. Em seguida, novos padrões são adicionados ao dicionário, juntamente com seus inversos. Além disso, a variável de total de padrões é incrementada, assim como a do tamanho dos mesmos, para calcular a estatística de tamanho dos padrões inseridos no dicionário.

Se ainda houver conteúdo no bloco para processar, o ciclo continua. Caso contrário, o último código para o último padrão "Pb" é enviado para a saída. As variáveis são então atualizadas para o próximo ciclo do loop, incluindo o índice (*ind*), que avança para o próximo conjunto de caracteres no bloco de conteúdo, e "Pa", que é atualizado para o último padrão "Pb" encontrado.

```

void compression(HashTable *hashTable, const std::vector<char> &block_content, unsigned long max_dic_size, std::string &output_buffer)
{
    int block_length = block_content.size();
    string Pa = string(1, block_content[0]);
    int ind = 2;
    int i = 1;
    int iteration = 1;
    cout << "Pa: " << Pa << endl;
    while (ind + i - 1 <= block_length)
    {
        int code = search(hashTable, Pa, max_dic_size);
        i = 1;
        string Pb = string(1, block_content[ind - 1]);
        cout << "Pb: " << Pb << endl;

        while (ind + i <= block_length)
        {
            string character = string(1, block_content[ind + i - 1]); // caractere a seguir ao pb
            if (search(hashTable, concat(Pb, character), max_dic_size))
            {
                // se encontrar,
                Pb = concat(Pb, character); // atualiza valor Pb e incrementa i
                i++; // anda à frente do ind (Pb) para ver qual o valor maior do buffer de entrada
                pattern_found++;
            }
            else
            {
                break;
            }
        }
        output(code, output_buffer);
        int j = 1;
        unsigned long t = max_dic_size - 1;
        while (j <= i && t < max_dic_size)
        {
            cout << "Padrão a ser inserido: " << concat(Pa, Pb.substr(0, j)) << endl;
            string current_pattern = concat(Pa, Pb.substr(0, j));
            t = insert(hashTable, code, concat(Pa, Pb.substr(0, j)), max_dic_size);
            total_pattern_count++;
            total_pattern_size += current_pattern.length();
            if (t < max_dic_size)
            {
                cout << "Padrão inverso a ser inserido: " << reverse(concat(Pa, Pb.substr(0, j))) << endl;
                t = insert(hashTable, code, reverse(concat(Pa, Pb.substr(0, j))), max_dic_size);
            }
            j++;
        }
        if (ind + i > block_length)
        {
            output(search(hashTable, Pb, max_dic_size), output_buffer);
            return;
        }
        else
        {
            ind += i;
            Pa = Pb;
            i = 1;
        }
        iteration++;
    }
}

```

Figura 4: Função de compressão.

3 Análise Crítica

O programa inicia ao receber um ficheiro, lendo-o e dividindo-o em blocos de acordo com o tamanho especificado pela linha de comando (função *read_block*). O tamanho máximo do dicionário é também especificado pela linha de comandos, de modo a implementar a otimização opcional. Cada bloco é então comprimido utilizando a função *compression*. Esta função usa as funções *concat* e *reverse* e retorna o código de Pa para o buffer de saída, o qual é guardado no ficheiro de saída "saída.txt" (função *save_to_file*). Durante a execução, o programa calcula estatísticas como o tempo total, o número de bytes processados, o número de padrões encontrados, o tamanho médio dos padrões e o número de resets do dicionário.

Algumas funções importantes de referir e analisar seriam a função *hash*, *insert* e *search*.

A função *hash_function* tem como funcionalidade calcular o índice na hash table para um padrão específico representado pela string value. Para isso, soma os valores ASCII de cada caractere na string e, em seguida, calcula o resto da divisão dessa soma pelo tamanho máximo do dicionário (*max_dic_size*). O valor resultante é, então, devolvido como o índice na tabela onde o padrão deve ser armazenado.

```
//Função para a função hash
int hash_function(const std::string &value, unsigned long max_dic_size)
{
    unsigned int hash = 0;
    for (char c : value)
    {
        hash += static_cast<unsigned int>(c); // Soma o valor ASCII de cada caracter
    }
    return hash % max_dic_size;
}
```

Figura 5: Função hash.

A função atual apenas soma os valores ASCII dos caracteres, o que pode resultar em colisões levando a uma distribuição desigual dos padrões, o que representa uma limitação.

A função *insert* inicialmente verifica se o dicionário está cheio, caso esteja a função *reset* é chamada para limpar o dicionário. Adicionalmente, faz a verificação da presença do padrão na hash table, para evitar duplicados. Em seguida, a função calcula o índice hash para o novo padrão usando a função *hash_function*. Se não houver nenhum item na posição da tabela correspondente ao índice calculado, o novo item é inserido nessa posição. Caso contrário, se já houver itens nessa posição, a função percorre a lista ligada naquela posição para encontrar o final e insere o novo item no final da lista, retornando o índice hash onde o padrão foi armazenado.


```

unsigned int insert(HashTable *hashtable, int key, const std::string &value, unsigned long max_dict_size)
{
    // Verificar se a tabela está cheia após os 256 padrões iniciais
    if (hashtable->count >= max_dict_size)
    {
        reset(hashtable, max_dict_size); //faz reset da tabela hash com as 256 posições iniciais
    }

    int index = hash_function(value, max_dict_size);
    // Verificar se o padrão já está presente
    if (search(hashtable, value, max_dict_size) != 0)
    {
        return 0;
    }

    Ht_item *item = new Ht_item; // Alocando memória para o novo item
    if (item == nullptr)
    {
        std::cerr << "Erro ao alocar memória para item" << std::endl;
        return 0;
    }
    item->key = key;
    item->value = value;
    item->next = nullptr;

    if (hashtable->items[index] == nullptr)
    {
        // Se não houver nenhum item no índice hash, insira o novo item
        item->key = hashtable->count; // o count começa em 1 e incrementa, key seja o número do padrão que estou a inserir
        hashtable->items[index] = item;
        hashtable->count++;
        return item->key; // Retorna o índice hash
    }
    else
    {
        Ht_item *current = hashtable->items[index];

        // Percorra a lista ligada para encontrar se o padrão já está presente
        while (current->next != nullptr)
        {
            if (current->value == value)
            {
                // Verifica se o padrão já está presente
                return 0; // Retorna zero se o padrão já estiver presente
            }
            current = current->next;
        }

        // Chegamos ao final da lista ligada sem encontrar o padrão, então vamos inseri-lo no final
        item->key = hashtable->count; // o count começa em 1 e incrementa, key seja o número do padrão que estou a inserir
        current->next = item;
        hashtable->count++;

        return item->key; //retorna o código
    }
}

```

Figura 6: Função de inserção de padrões.

Para exemplificar o funcionamento da função insert, vamos utilizar o exemplo de ficheiro fornecido no enunciado e demonstrar através de um diagrama como ela opera.

Quando o padrão ABA, descoberto na função de compressão, é inserido no dicionário com o código correspondente 262, a função insert calcula e verifica que a posição hash é 196. O dicionário é inicializado com 256 posições iniciais, e essa posição já se encontra preenchida. Assim, a função acessa à posição, verifica que está preenchida e começa a percorrer a lista ligada, adicionando os itens sempre no final.

Podemos comprovar este comportamento, ao usar a função *display_table*:

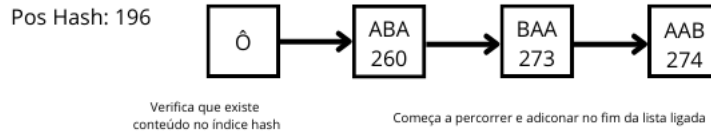


Figura 7: Diagrama de funcionamento

```

Hash: 192
  Padrão: 0 Código: 192
Hash: 193
  Padrão: 0 Código: 193
Hash: 194
  Padrão: 0 Código: 194
Hash: 195
  Padrão: 0 Código: 195
Hash: 196
  Padrão: 0 Código: 196
  Padrão: ABA Código: 260
  Padrão: BAA Código: 273
  Padrão: AAB Código: 274
Hash: 197
  Padrão: 0 Código: 197
  Padrão: BAB Código: 259
  Padrão: ABB Código: 263
  Padrão: BBA Código: 264
Hash: 198
  Padrão: 0 Código: 198
Hash: 199
  Padrão: 0 Código: 199
Hash: 200
  Padrão: 0 Código: 200
  
```

Figura 8: Resultado da função para imprimir o dicionário.

A função `search` calcula o índice hash para o padrão fornecido usando a função `hash_function`. Em seguida, verifica se há algum item na posição da tabela correspondente ao índice calculado. Se houver itens nessa posição, a função percorre a lista ligada para encontrar o padrão. Se o padrão for encontrado, retorna o código associado a esse padrão. Se o padrão não for encontrado, a função retorna zero.

```

//Função para procurar um padrão na tabela hash
unsigned int search(HashTable *hashtable, const std::string &value, unsigned long max_dict_size)
{
    int index = hash_function(value, max_dict_size); // Calcula o índice hash do padrão

    // Verifica se há algum item na posição da tabela hash correspondente ao índice calculado
    Ht_item *current = hashtable->items[index];
    while (current != nullptr)
    {
        // Verifica se o valor do item é igual ao padrão
        if (current->value == value)
        {
            // cout << "Padrão encontrado: " << value << ", Código: " << current->key << endl;
            return current->key; // Retorna o código do padrão se encontrado
        }
        current = current->next; // Avança para o próximo item na lista ligada
    }

    // Se o padrão não for encontrado, retorna zero
    //cout << "Padrão não encontrado: " << value << endl;
    return 0;
}
  
```

Figura 9: Função de procura de padrões.

4 Análise de testes

Para avaliar o desempenho do projeto desenvolvido, vamos utilizar diferentes ficheiros de teste de vários tamanhos e tipos (ficheiros de texto, imagem, PDF, etc), e avaliar as métricas de tempo de processamento e o número de blocos.

4.1 Ficheiros de texto

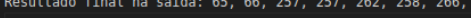
Inicialmente, vamos demonstrar o resultado do nosso algoritmo utilizando o ficheiro de texto dado no exemplo do enunciado, para comprovar que a função *compression* funciona de acordo com o solicitado.

```

C:\militar>cmd /k "cd Desktop\SRANS & ./lz4wr 65536 65536
Autor: Camila Pinto, Eduardo Dinis, Barbara Fonseca
Data de criação: 15/04/2024
Nome do ficheiro de entrada: ola.txt
Nome do ficheiro de saída: saida.txt
Tamanho máximo do dicionário: 65536
Tipo de dicionário inicial: HashTable, inicializada com 256 padrões.
Tipo de limpeza: Reset do dicionário, mantendo apenas as 256 entradas padrão.
Pa: A
Pb: B
Padrão a ser inserido: AB
Padrão inverso a ser inserido: BA
Pb: A
Padrão a ser inserido: BA
Padrão inverso a ser inserido: AB
Padrão a ser inserido: BAB
Padrão inverso a ser inserido: BAB
Pb: A
Padrão a ser inserido: ABA
Padrão inverso a ser inserido: ABA
Padrão a ser inserido: ABAB
Padrão inverso a ser inserido: BABA
Pb: B
Padrão a ser inserido: ABB
Padrão inverso a ser inserido: BBA
Padrão a ser inserido: ABBA
Padrão inverso a ser inserido: ABBA
Padrão a ser inserido: ABBA
Padrão inverso a ser inserido: BABBA
Padrão a ser inserido: ABBA
Padrão inverso a ser inserido: ABABBA
Pb: B
Padrão a ser inserido: BABAB
Padrão inverso a ser inserido: BABAB
Padrão a ser inserido: BABABA
Padrão inverso a ser inserido: ABABAB
Pb: A
Padrão a ser inserido: BAA
Padrão inverso a ser inserido: AAB
Padrão a ser inserido: BAAB
Padrão inverso a ser inserido: BAAB
Padrão a ser inserido: BAABB
Padrão inverso a ser inserido: BBAAB
Padrão a ser inserido: BAABBA
Padrão inverso a ser inserido: ABBAAB
Padrão a ser inserido: BAABBA
Padrão inverso a ser inserido: BABBAAB
Pb: B
Padrão a ser inserido: ABABAB
Padrão inverso a ser inserido: BBABBA
Padrão a ser inserido: ABABBA
Padrão inverso a ser inserido: ABABBA
Padrão a ser inserido: ABBAAB
Padrão inverso a ser inserido: BABBAAB
Padrão a ser inserido: ABBAABBA
Padrão inverso a ser inserido: ABABABBA
Tempo total da operação : 0 segundos
*****Estatísticas : *****
Número total de bytes processados: 21 bytes
Número de padrões encontrados: 13
Número de código de padrão enviado para o output: 8
Tamanho médio dos padrões inseridos no dicionário: 4.9
Número de resets do dicionário: 0
Tamanho do bloco 1: 21 bytes
Número de blocos processados: 1

```

Figura 10: Resultado do programa para a sequência: ABABABBABABAABBABBABA.



Terminal Help

C++ lzwdr_main.cpp | **saida.txt** x | testetexto2.txt

saida.txt

```
1 Resultado final na saída: 65, 66, 257, 257, 262, 258, 266, 262, 0*
2
```

Figura 11: Ficheiro de saída

Comparando com o resultado demonstrado no enunciado, o nosso algoritmo está funcional e a calcular corretamente a procura dos padrões para compressão. Relativamente ao tempo, para arquivos muito pequenos, o tempo inicial para preparar e inicializar o algoritmo pode ser insignificante em comparação com o tempo de compressão real. Adicionalmente, na Figura 11, verificámos que o código de saída está a ser bem armazenado no ficheiro e na Figura 10 verificamos que estão presentes os prints solicitados.

Aumentando agora o ficheiro de texto para 204,8 kBytes, vamos analisar o resultado:

```
Padrão inverso a ser inserido: rrcd
Tempo total da operação : 6 segundos
*****Estatísticas : *****
Número total de bytes processados: 204800 bytes
Número de padrões encontrados: 109634
Número de código de padrão enviado para o output: 95166
Tamanho médio dos padrões inseridos no dicionário: 3.7735
Número de resets do dicionário: 6
Tamanho do bloco 1: 65536 bytes
Tamanho do bloco 2: 65536 bytes
Tamanho do bloco 3: 65536 bytes
Tamanho do bloco 4: 8192 bytes
Número de blocos processados: 4
```

Figura 12: Resultado do programa para um ficheiro maior.

Através da Figura 11, é possível analisar que a divisão dos blocos está a ser bem feita para os 204800 bytes de entrada (tamanho do ficheiro). No entanto, ao aumentarmos o tamanho do ficheiro, notámos um tempo de compressão de 6 segundos. Este aumento pode ser atribuído à complexidade adicional de processamento necessária para lidar com um volume maior de dados.

4.2 PDF

No ficheiro PDF, a redundância refere-se à repetição de padrões de dados semelhantes ou idênticos ao longo do documento. Sendo o ficheiro em PDF comprimido, isso significa que muitos desses padrões repetitivos foram compactados num formato mais eficiente. No entanto, durante a compressão adicional usando o algoritmo LZWdR, os padrões que restam podem ser muito curtos, o que significa que são inseridos padrões curtos. Esta situação aumenta a redundância nos dados e torna a compressão menos eficaz. Para testar, utilizámos o enunciado do trabalho prático, com 229,3 kBytes. Este gerou quatro blocos de dados e demorou 11 segundos, uma diferença de 5 segundos em relação ao ficheiro de texto, que é 24,5 kBytes mais pequeno. Assim, o PDF demora mais tempo para ser comprimido. Adicionalmente, comparando com o exemplo da sequência, Figura 9, o tamanho médio dos padrões inseridos é mais curto, o que demonstra que estão a ser inseridos padrões mais curtos.

```
Tempo total da operação : 11 segundos
*****Estatísticas : *****
Número total de bytes processados: 229254 bytes
Número de padrões encontrados: 85354
Número de código de padrão enviado para o output: 143900
Tamanho médio dos padrões inseridos no dicionário: 4.17412
Número de resets do dicionário: 6
Tamanho do bloco 1: 65536 bytes
Tamanho do bloco 2: 65536 bytes
Tamanho do bloco 3: 65536 bytes
Tamanho do bloco 4: 32646 bytes
Número de blocos processados: 4
camila@camila:~/Desktop/SRAMs
```

Figura 13: Resultado do programa para um ficheiro pdf.

4.3 Imagem

O princípio do PDF também se aplica a ficheiros de imagem. Para testar um ficheiro de imagem (.jpeg), utilizámos uma imagem com 257,0 kBytes. Neste caso, o tamanho do ficheiro é maior e o tempo de processamento é de 16 segundos. Além disso, ao analisarmos as estatísticas, notamos que o tamanho médio dos padrões inseridos no dicionário é inferior, o que indica que estão a ser inseridos padrões mais curtos.

```
Padrao inverso a ser inserido: 00
Tempo total da operação : 16 segundos
*****Estatísticas : *****
Número total de bytes processados: 257011 bytes
Número de padrões encontrados: 91455
Número de código de padrão enviado para o output: 165556
Tamanho médio dos padrões inseridos no dicionário: 3.00984
Número de resets do dicionário: 7
Tamanho do bloco 1: 65536 bytes
Tamanho do bloco 2: 65536 bytes
Tamanho do bloco 3: 65536 bytes
Tamanho do bloco 4: 60403 bytes
Número de blocos processados: 4
camila@camila:~/Desktop/SRAM$
```

Figura 14: Resultado do programa com uma imagem.

5 Manual de Utilização

Para compilar:

- make clean all

Para testar:

- ./lzwdr [tamanhobloco] [tamanhomaxdicionario]

6 Conclusão

De uma maneira geral, consideramos que implementamos com sucesso o algoritmo de compressão proposto, utilizando a estrutura de hashtable para o dicionário e todas as funções derivadas. Implementamos todas as funcionalidades obrigatórias e, adicionalmente, permitimos a otimização do tamanho dos blocos e do tamanho máximo do dicionário, valorizamos a limpeza do dicionário e fornecemos informações adicionais sobre os padrões encontrados e inseridos. No entanto, o nosso algoritmo não se mostra eficiente com ficheiros de áudio, uma vez que demora demasiado tempo e nunca conclui o processo.

Como trabalho futuro, poderíamos otimizar mais eficientemente o algoritmo de compressão para melhorar os tempos obtidos, utilizar a função de hash DJB2 para gerar uma dispersão mais uniforme dos dados e implementar outras estatísticas mencionadas.

Referências

- [1] L. Fitriya, T. Purboyo, and A. Prasasti, “A review of data compression techniques,” *International Journal of Applied Engineering Research*, vol. 12, pp. 8956–8963, Jan. 2017.
- [2] “Codificação por padrões.” [Online]. Available: https://elearning.uminho.pt/ultra/courses/_59452_1/cl/outline
- [3] “Trabalho prático nº1 – compressão lzwdr.” [Online]. Available: https://elearning.uminho.pt/ultra/courses/_59452_1/cl/outline
- [4] “Hash Table vs Trie,” Dec. 2022, section: DSA. [Online]. Available: <https://www.geeksforgeeks.org/hash-table-vs-trie/>