Implementação de Algoritmo de Compressão LZW

Carlos Vinícius Costa Neves (20180028631)
Rafael Sobral de Morais (20180033515)

16/11/2022

1 Introdução

O algoritmo LZW é uma técnica de compressão de arquivos binários [2, 3]. Ao contrário do código de Huffman, que visa associar um único código prefixo a cada caracter da mensagem original de maneira ótima, o LZW aproveita-se da repetição de padrões (i.e., sequências de caracteres que ocorrem de forma recorrente no arquivo). Em suma, o algoritmo mantém armazenado um "dicionário" que contém códigos únicos para cada símbolo do alfabeto do arquivo original. Além disso, o dicionário também contém códigos para sequências que ocorrem no arquivo. Sequências promissoras e que, portanto, devem ser armazenadas no dicionário, são escolhidas de forma heurística.

Neste trabalho, uma implementação do algoritmo de compressão (e descompressão) LZW na linguagem C++ é apresentada. As próximas seções do trabalho apresentam as escolhas tomadas em relação aos detalhes de implementação, bem como os resultados obtidos para os arquivos binários providos pelo professor.

2 Implementação do algoritmo

O trecho de código a seguir apresenta a estrutura `LZW`, que contém todas as informações necessárias para implementar o algoritmo. Na estrutura, tanto `dict` quanto `inv_dict` são usados como o dicionário. O primeiro serve para a etapa de compressão, e usa como chave um `string` que representa o padrão a ser procurado. O segundo, por sua vez, é usado na descompressão, e faz uso de uma lógica semelhante. O inteiro `max_word_size` representa o tamanho em *bits* K dos códigos do dicionário. Por fim, os métodos `Encode` e `Decode` contém, respectivamente, os algoritmos de compressão e descompressão.

```
1  struct LZW {
2    map<string, uint16_t> dict;
3    map<uint16_t, string> inv_dict;
```

```
4     int max_word_size;
5
6     void FillDictWithAlphabet()
7     {
8       for (int i = 0; i < 256; i++)
9       {
10        string tmp;
11        tmp += (char) i;
12
13        dict[tmp] = i;
14        inv_dict[i] = tmp;
15      }
16    }
17
18    LZW() : max_word_size(8) { FillDictWithAlphabet(); }
19    LZW(int k) :  max_word_size(k) { FillDictWithAlphabet(); }
20
21    void Encode(string input_filename, string output_filename);
22    void Decode(string input_filename, string output_filename);
23  };
```

A versão canônica do algoritmo (vista em aula) foi implementada. Para limitar o tamanho do dicionário, utilizou-se o parâmetro K de forma estática, i.e., quando o dicionário atinge o tamanho final 2^K, impede-se que novos padrões e códigos sejam adicionados. Além disso, optou-se por adicionar o valor K no cabeçalho do arquivo comprimido, já que dessa forma (*i*) ele não precisa ser informado durante a descompressão; e (*ii*) o impacto na razão de compressão é mínimo (o arquivo final sofre o acréscimo de um *byte*).

2.1 Compressão

Os métodos `Encode` e `Decode` são extensos, e portanto, serão omitidos do relatório, e podem ser encontrados no repositório do *Github* associado ao projeto. Apenas partes relevantes do código serão apresentadas.

O trecho de código a seguir, que é parte do método `Encode`, mostra como os *bytes* resultantes da etapa de compressão são escritos no arquivo de saída `output_file`. O *loop* na linha 3 itera por todos os *bits* do código corrente, que possui tamanho `max_word_size`. O bit a ser escrito no arquivo é determinado por meio de operações *bitwise* na linha 5. Na linha 7, checa-se se o *byte* está pronto para ser escrito no arquivo através do inteiro `bits_written`, que informa quantos *bits* foram lidos. Se for o caso, o *byte* a ser escrito é resetado (linhas 13–14). Por fim, a cada etapa do *loop*, os *bits* do *byte* atual são movidos para a esquerda também através de operações *bitwise* (linhas 17–18).

```
1  /* Output bits */
2  bool byte_written = false;
3  for (int bit = max_word_size - 1; bit >= 0; bit--)
```

```
4  {
5    int out_bit = ((curr_keyword >> bit) & 1);
6
7    if (bits_written == 8)
8    {
9      output_file.write((const char*) &byte_to_write, sizeof(char));
10
11     byte_written = true;
12
13     byte_to_write = 0;
14     bits_written = 0;
15   }
16
17   byte_to_write <<= 1;
18   byte_to_write |= out_bit;
19   bits_written++;
20 }
```

O trecho de código a seguir mostra como ocorre a verificação de tamanho do dicionário. O tamanho máximo permitido é calculado por meio de um *shift* de tamanho K (`max_word_size`). A *string* `curr_str` contém o padrão (sequência de caracteres) corrente.

2.2 Descompressão

A etapa de descompressão ocorre de forma muito semelhante à de compressão. Dessa vez, no entanto, o arquivo de entrada é lido um *byte* por vez, e o próximo código a ser escrito (que possui exatamente K *bits*) deve ser determinado sempre que K *bits* são lidos. Isso é ilustrado no trecho de código a seguir, no qual os *bits* do *byte* corrente são lidos através de um *loop* e operações de *shift* (linhas 38–41) assim como no método `Encode`. Quando K *bits* são lidos e um código é obtido (linha 3), cada um dos possíveis casos (e.g., o código já existe no dicionário, o código ainda não existe) é contemplado. Observe que no caso específico em que o código lido é o primeiro (linha 7), apenas escreve-se o símbolo correspondente diretamente no arquivo de saída. Os casos mais elaborados são tratados nas linhas 12–32, exatamente como visto em aula.

```
1  for (int bit = 8 - 1; bit >= 0; bit--)
2  {
3    if (bits_read == max_word_size)
4    {
5      keywords_read++;
6
7      if (keywords_read == 1)
8      {
9        prev_keyword = curr_keyword;
10       output_file.write((const char*) &inv_dict[curr_keyword][0], sizeof(
             char));
```

```
11      }
12      else
13      {
14        string prev_str = inv_dict[prev_keyword];
15
16        if (!inv_dict.count(curr_keyword))
17        {
18          string temp = prev_str + prev_str[0];
19          if ( inv_dict.size() < (1 << max_word_size) )
20            inv_dict[(uint16_t) inv_dict.size()] = temp;
21          output_file.write((const char*) temp.data(), temp.size());
22        }
23        else
24        {
25          string temp = prev_str + inv_dict[curr_keyword][0];
26          if ( inv_dict.size() < (1 << max_word_size) )
27            inv_dict[(uint16_t) inv_dict.size()] = temp;
28          output_file.write((const char*) inv_dict[curr_keyword].data(),
                inv_dict[curr_keyword].size());
29        }
30
31        prev_keyword = curr_keyword;
32      }
33
34        curr_keyword = 0;
35        bits_read = 0;
36    }
37
38    curr_keyword <<= 1;
39    curr_keyword |= ((curr_byte >> bit) & 1);
40
41    bits_read++;
42  }
```

3 Resultados e discussão

A implementação pode ser encontrada no seguinte *link*: `https://github.com/cvneves/ITI`. O algoritmo LZW foi implementado na linguagem C++. Nenhuma biblioteca ou código adicional foi utilizada para escrever ou ler os arquivos. O código foi testado em um arquivo de texto e um arquivo de vídeo com uma faixa de valores de K especificada pelo professor. Mais especificamente, a compressão foi realizada para cada um dos dois arquivos utilizando-se $K = 9, \ldots, 16$, gerando um total de 32 arquivos comprimidos (16 para o arquivo de texto e 16 para o arquivo de vídeo).

O código funciona com dois executáveis (um para a compressão, e outro para a descompressão). A seguinte sequência de comandos, que deve ser executada a partir da pasta raiz do repositório, mostra como gerar os executáveis utilizando-se

K	Tamanho	Tempo	#Índices	RC	RC2
9	10270637	19.53	512	1.60	28,510.08
10	9251224	20.72	1024	1.78	12,829.54
11	8402307	21.42	2048	1.95	5,831.61
12	7965064	21.74	4096	2.06	2,672.82
13	7642690	22.42	8192	2.15	1,233.61
14	7330312	23.33	16384	2.24	572.75
15	7039895	25.68	32768	2.33	267.28
16	6758967	25.94	65536	2.43	125.29

Tabela 1: Dados da compressão do corpus de texto.

K	Tamanho	Tempo	#Índices	RC	RC2
9	2359913	2.84	512	0.89	3,665.01
10	2613379	3.02	1024	0.81	1,649.26
11	2862105	3.30	2048	0.74	749.66
12	3104068	3.49	4096	0.68	343.59
13	3183844	3.86	8192	0.66	158.58
14	3102088	3.93	16384	0.68	73.63
15	2850723	3.90	32768	0.74	34.36
16	2579157	3.97	65536	0.82	16.11

Tabela 2: Dados da compressão do arquivo de vídeo.

a ferramente *CMake* e como comprimir e descomprimir um arquivo.

```
 mkdir build && cd build && cmake .. && cd ..
./encode <arquivo_original> <arquivo_comprimido> <K>
./decode <arquivo_comprimido> <arquivo_descomprimido>
```

Os resultados resumidos se encontram nas tabelas 1 e 2. Nas tabelas, a coluna **K** indica o valor de K, a coluna **Tamanho** indica o tamanho do arquivo comprimido, a coluna **#Índices** informa o tamanho do dicionário ao final da execução, e as colunas **RC** e **RC2** informam, respectivamente, os valores para a razão de compressão obtidos através das equações (1) e (2). De início, inspecionando-se as tabelas, percebe-se que os dicionários atingiram o tamanho máximo 2^K em todas as execuções do algoritmo.

$$RC = \frac{tamArgOriginal}{tamArqComprimido} \tag{1}$$

$$RC2 = \frac{tamArgOriginal}{\frac{K}{8} totalIndices} \tag{2}$$

Os resultados podem ser melhor observados através das figuras 1–3, que apresentam, respectivamente, as razões de compressão conforme as equações (1)–(2) e

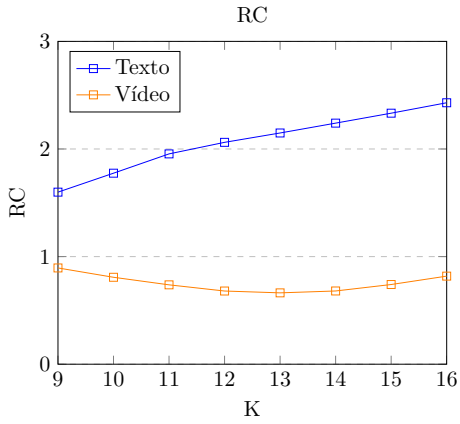

Figura 1: Razão de compressão (equação 1)

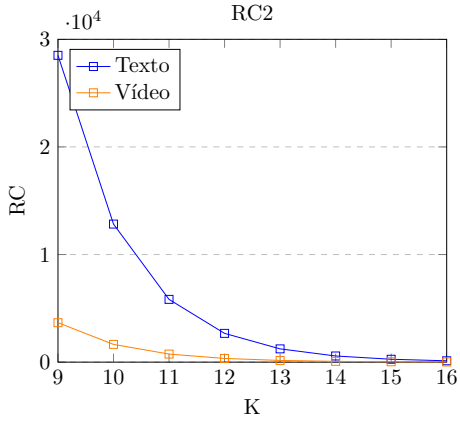

Figura 2: Razão de compressão (equação 2)

os tempos execução. No caso do arquivo de texto, é possível observar na Figura 1 que a razão de compressão aumenta conforme o valor de K aumenta. No caso do arquivo de vídeo, no entanto, a razão é sempre menor que 1, e pouco varia. Esse resultado é esperado, pois além de um alfabeto reduzido, o arquivo de texto possui vários padrões (e.g., palavras, frases) que se repetem de forma recorrente. O arquivo de vídeo, por sua vez, já está comprimido, e provavelmente possui uma entropia muito alta, dificultando que seu tamanho seja diminuído significativamente através de compressões sem perda. A Figura 2 também apresenta um comportamento esperado de decrescimento rápido para as razões de compressão, já que os valores de tamanho máximo do dicionário aumentam de maneira exponencial em função de K. Por fim, conforme ilustrado na Figura 3, o valor de K possui uma influência significativa no tempo de execução no caso do arquivo de texto, e negligenciável no caso do arquivo de vídeo. Percebe-se, ainda, que o tamanho do arquivo a ser comprimido possui bastante impacto no tempo de execução.

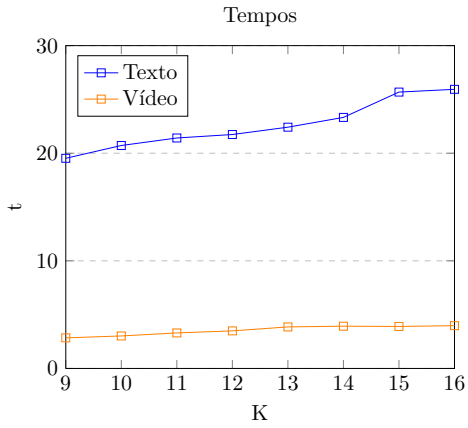

Figura 3: Tempos

A descompressão de arquivos também foi testada usando-se diferentes valores de K. Em todos os experimentos usando-se a ferramenta `diff`, o arquivo descomprimido gerado é exatamente igual ao original. Isso ocorreu tanto no caso do arquivo de texto quanto no arquivo de vídeo, que continuou reproduzível após a descompressão. Além disso, já que os códigos foram escritos *bit* a *bit* durante a compressão, os arquivos comprimidos acabaram adquirindo o tamanho esperado, utilizando-se sempre K *bits* para armazenar cada código.

4 Considerações finais

Neste trabalho, uma implementação do algoritmo LZW foi apresentada. O algoritmo pôde ser testado com diferentes tipos de arquivos e tamanhos de código. Algumas dificuldades foram encontradas no decorrer da implementação, sobretudo na escrita *bit* a *bit* no arquivo. Contudo, no geral, o algoritmo se comportou como previsto pela teoria vista em aula, conforme mostram os experimentos e resultados apresentados.

Para trabalhos futuros, sugere-se melhorias ao método `Encode`, como o uso de um par de inteiros para representar a chave da estrutura `dict` em vez de uma *string*, como explicado por [1]. Além disso, pode-se explorar maneiras mais eficientes de geração dos *bytes* a serem escritos no arquivo por meio de operações *bitwise*. Tais modificações podem diminuir significativamente o tempo de execução do algoritmo.

Referências

[1] Lempel-Ziv-Welch (LZW) Encoding Discussion and Implementation, `http://michael.dipperstein.com/lzw/`

[2] LZW Data Compression
`https://marknelson.us/posts/1989/10/01/lzw-data-compression.html`

[3] LZW Data Compression Revisited
`https://marknelson.us/posts/2011/11/08/lzw-revisited.html`