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. 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 inteiro `max_word_size` representa o número de *bits* K dos códigos do dicionário. O segundo, por sua vez, é usado na descompressão, e faz uso de uma lógica semelhante. 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 (vista em aula) do algoritmo 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 inseridos.

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 14–15). Por fim, a cada etapa do *loop*, os *bits* do *byte* atual são movidos para a esquerda também através de operações *bitwise* (linhas 19–20).

```
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.

```
1  if(!dict.count(curr_str) && dict.size() < (1 << max_word_size))
2          dict[curr_str] = dict.size();
```

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 novo código (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 existe, o código ainda não existe) é contemplado.

```
1  input_file.read((char*) &max_word_size, sizeof(char));
```

```
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

1. C++
   2. a. Sim b. Não
   c.
   d. Tá igual
   3. Sim
   4. Operadores *bitwise*
   5.
   6. Todos

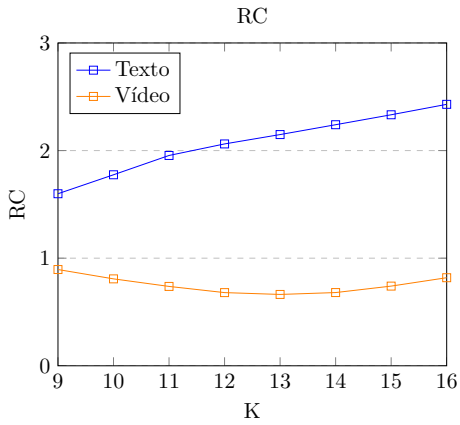

Figura 1: RC

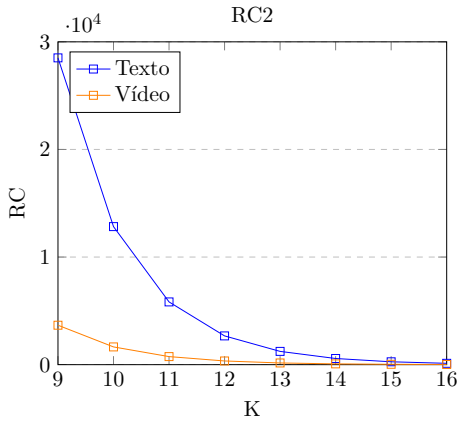

Figura 2: RC

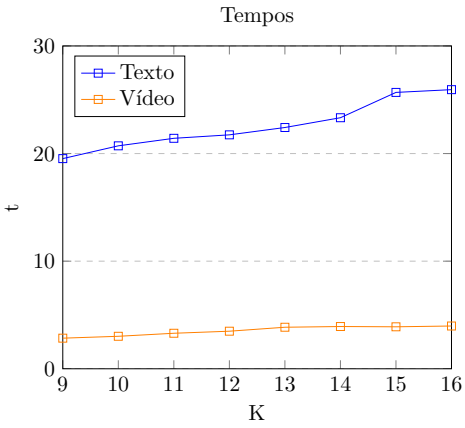

Figura 3: Tempos

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.