**Universidade Federal de Alfenas - UNIFAL-MG**
Bacharelado em Ciência da Computação
Prof. Luiz Eduardo da Silva

# Trabalho de Processamento de Imagens
## Atividade 2 - Compressão/descompressão de imagem

# Objetivo

O objetivo desse atividade é explorar os conceitos de compressão e descompressão usados nos formatos de imagens. Especificamente, o intuito é desenvolver um descompactador para um formato que mescla a compactação LZW e a representação binária usando Base64.

# Problema

**Base64**

Base64[1] é um método para codificação de dados para transferência na Internet (codificação MIME para transferência de conteúdo). É utilizado frequentemente para transmitir dados binários por meios de transmissão que lidam apenas com texto, como por exemplo para enviar arquivos anexos por e-mail.

É constituído por 64 caracteres ([A-Z],[a-z],[0-9], "/"e "+") que deram origem ao seu nome. O carácter "="é utilizado como um sufixo especial e a especificação original (RFC 989) definiu que o símbolo "*"pode ser utilizado para delimitar dados convertidos, mas não criptografados, dentro de um stream.

A tabela de equivalência entre os caracteres e grupos de 6 bits utilizados para codificação Base64:

| car | bits | car | bits | car | bits | car | bits |
|---|---|---|---|---|---|---|---|
| A | 000000 | Q | 010000 | g | 100000 | w | 110000 |
| B | 000001 | R | 010001 | h | 100001 | x | 110001 |
| C | 000010 | S | 010010 | i | 100010 | y | 110010 |
| D | 000011 | T | 010011 | j | 100011 | z | 110011 |
| E | 000100 | U | 010100 | k | 100100 | 0 | 110100 |
| F | 000101 | V | 010101 | l | 100101 | 1 | 110101 |
| G | 000110 | W | 010110 | m | 100110 | 2 | 110110 |
| H | 000111 | X | 010111 | n | 100111 | 3 | 110111 |
| I | 001000 | Y | 011000 | o | 101000 | 4 | 111000 |
| J | 001001 | Z | 011001 | p | 101001 | 5 | 111001 |
| K | 001010 | a | 011010 | q | 101010 | 6 | 111010 |
| L | 001011 | b | 011011 | r | 101011 | 7 | 111011 |
| M | 001100 | c | 011100 | s | 101100 | 8 | 111100 |
| N | 001101 | d | 011101 | t | 101101 | 9 | 111101 |
| O | 001110 | e | 011110 | u | 101110 | + | 111110 |
| P | 001111 | f | 011111 | v | 101111 | / | 111111 |

[1] https://pt.wikipedia.org/wiki/Base64

**LZW**

*"A técnica, chamada codificação de Lempel-Ziv-Welch (LZW), atribui palavras código de tamanho fixo a sequências de símbolos fonte de tamanho variável. [...] Uma importante característica da codificação LZW é que ela não requer conhecimento antecipado da probabilidade de ocorrência dos símbolos que serão codificados".* [Gonzalez 2010]

Codificação de LZW.

Conceitualmente, a codificação de LZW é bastante simples [Welch 1984]. No início do processo de codificação, é construído um banco de códigos ou dicionário contendo os símbolos fonte a serem codificados. Para imagens monocromáticas de 8 bits, as 256 primeiras palavras do dicionário são atribuídas às intensidades 0, 1, 2, ..., 255. À medida que o codificador analisa sequencialmente os pixels da imagem, as sequências de intensidade que não estão contidas no dicionário são incluídas na próxima localização não utilizada do dicionário. Se os dois primeiros pixels da imagem forem brancos, por exemplo, a sequência "255-255"pode ser atribuída à posição 256, com o endereço seguindo as posições reservadas para os níveis de intensidade 0 a 255. Da próxima vez que dois pixels brancos consecutivos forem encontrados, a palavra código 256, o endereço da posição contendo a 255-255 é utilizada para representá-los. Se um dicionário de 9 bits e 512 palavras for empregado no processo de codificação, os (8 + 8) bits originais que foram utilizados para representar os dois pixels 255, do exemplo anterior, são substituídos por uma única palavra código de 9 bits. Claramente, o tamanho do dicionário é um importante parâmetro do sistema. Se for pequeno demais, a detecção de sequências de nível de intensidade correspondente será menos provável; se for grande demais, o tamanho das palavras código afetará desfavoravelmente o desempenho da compressão.

Exemplo:

Considere a imagem de uma borda vertical, de 8 bits e com dimensões 4 x 4 pixels:

39 39 126 126

39 39 126 126

39 39 126 126

39 39 126 126

A Tabela 2 detalha os passos envolvidos na codificação de seus 16 pixels. Um dicionário de 512 palavras com o seguinte conteúdo inicial é presumido:

| Posição | 0 | 1 | 2 | ... | 255 | 256 | 257 | ... | 511 |
|---|---|---|---|---|---|---|---|---|---|
| Valor | 0 | 1 | 2 | ... | 255 | - | - | ... | - |

Tabela 1: Dicionário inicial

As posições 256 a 511 inicialmente não são utilizadas.

A Imagem é codificada por meio do processamento de seus pixels da esquerda para a direita e de cima para baixo. Cada valor de intensidade sucessivo é concatenado com uma variável - coluna 1, da Tabela 2 - chamada "sequência atualmente reconhecida". Como podemos ver essa variável é inicialmente nula ou vazia . É realizada uma busca no dicionário para cada sequência concatenada e, se encontrada, como foi o caso na primeira linha da tabela, ela é substituída pela sequência recém-concatenada e reconhecida (isto é localizada no dicionário). Isso foi feito na coluna 1 da linha 2. Nenhum código de saída é gerado e o dicionário não é alterado. Se a sequência concatenada não for encontrada contudo, o endereço da sequência atualmente reconhecida é produzida como o próximo valor codificado, a sequência concatenada, porém não reconhecida, é incluída no dicionário e a sequência atualmente reconhecida é inicializada no valor do pixel atual. Isso ocorreu na linha 2 da tabela. As duas últimas colunas detalham as sequências de intensidade adicionadas ao dicionário ao varrer toda a imagem. Nove palavras código adicionais são definidas. Na conclusão da codificação, o dicionário contém 265 palavras código e o algoritmo LZW identificou com sucesso várias sequências de intensidade repetidas

- reduzindo a imagem original de 128 bits para 90 bits (isto é, 10 códigos de 9 bits). A saída codificada é obtida lendo a terceira coluna de cima para baixo. A taxa de compressão resultante é 1,42:1.

| Sequência atualmente reconhecida | Pixel sendo processado | Saída codificada | Posição no dicionário (palavra código) | Entrada no dicionário |
|---|---|---|---|---|
| | 39 | | | |
| 39 | 39 | 39 | 256 | 39-39 |
| 39 | 126 | 39 | 257 | 39-126 |
| 126 | 126 | 126 | 258 | 126-126 |
| 126 | 39 | 126 | 259 | 126-39 |
| 39 | 39 | | | |
| 39-39 | 126 | 256 | 260 | 39-39-126 |
| 126 | 126 | | | |
| 126-126 | 39 | 258 | 261 | 126-126-39 |
| 39 | 39 | | | |
| 39-39 | 126 | | | |
| 39-39-126 | 126 | 260 | 262 | 39-39-126-126 |
| 126 | 39 | | | |
| 126-39 | 39 | 259 | 263 | 126-39-39 |
| 39 | 126 | | | |
| 39-126 | 126 | 257 | 264 | 39-126-126 |
| 126 | | 126 | | |

Tabela 2: Acompanhamento de uma compressão

As listagens 1 e 2 na seção de anexos neste documento apresentam implementações para os algoritmos de compressão e descompressão LZW, conforme discutido.

# Descrição

1. Nessa atividade você deverá implementar o descompactador de uma imagem codificada usando a compactação LZW com códigos de 12 bits e a representação dos códigos usando Base64, conforme apresentado anteriormente neste documento.

2. O arquivo a ser descompactado é um arquivo em formato texto com a extensão ".l64"e os seguintes campos:

   - A primeira linha contém o texto lzw-b64 (número mágico).
   - A segunda linha contém as dimensões da imagem (nl = número de linhas e nc = número de colunas, respectivamente)
   - As próximas linhas contém a codificação base64 dos códigos LZW dos (nl * nc) pixels da imagem. Para facilitar, os códigos LZW são de 12 bits. Assim, cada código LZW é traduzido para dois caracteres em Base 64 no arquivo compactado.

   Exemplo:

```
lzw-b64
279 213
BHA7BRBCBCAxBCBMA1A7BHBHA+BHBMBRBMBMEAEIBWBHBCA7EXEUBCEUBHBTBWEOBZBMA2BcBcBhBmBdBg
```

```
EWEQBRA1BMBTBmBMA7EwEPAxEQBMEeBAEBBTEWENBHEPBZEvA7A2EDA2EwBAEpBcEeBWEdBdBMBdBwBlBt
BmBZBZBtBuBtBZBhBfBMBgECBTBhBMFIBcBZEPEPEeBlBRBfBlBZFiBmBmB9B3B0BoBhBuBhFzB4BmBoBR
B2BZByBTBRFYE/ByBfB0B4BRBuBmGHB2A+EPF4BhBoGHF1EvFzGPGFBRFtB3BuEkFKGZFeEJBRBhEkBRBc
```

3. Por exemplo, a imagem .PGM com o seguinte conteúdo:

```
P2
# CREATOR: Image Processing using C-Ansi - ByDu
4  4
255
39  39 126 126  39  39 126 126  39  39 126 126  39  39 126 126
```

Está codificada nesse formato como:

```
lzw-b64
4 4
AnAnB+B+EAECEEEDEBB+
```

Observe que os dois primeiros caracteres em Base64 no arquivo compactado (A = 000000 e n = 100111), como um único código de 12 bits corresponde a 000000100111, que é igual ao valor 39. Outro exemplo, EA = 000100000000 é igual ao código 256.

# Entrega

1. Incluir um comentário no cabeçalho de cada programa fonte com o seguinte formato:

```
/*-------------------------------------------------------------------
*            UNIFAL - Universidade Federal de Alfenas.
*              BACHARELADO EM CIENCIA DA COMPUTACAO.
* Trabalho..: Descompactador do formato LZW-Base64
* Disciplina: Processamento de Imagens
* Professor.: Luiz Eduardo da Silva
* Aluno.....: Fulano da Silva
* Data......: 99/99/9999
*-------------------------------------------------------------------*/
```

2. O projeto deverá incluir um arquivo MAKEFILE para construção do descompactador, conforme código exemplo "negativo.zip", disponível no Moodle.

3. O programa deverá ser chamado em linha de comando da seguinte forma:

```
./decode <nome-arquivo-imagem>[.l64]
```

onde a extensão do arquivo de imagem (.l64) pode ser opcional.

4. O programa deverá gerar o arquivo .PGM, com mesmo nome do arquivo original.

5. Enviar num arquivo único (.ZIP), com todos os arquivos fonte do projeto através do Envio de Arquivo do MOODLE.

# Anexos

Listagem 1: Implementação da compressão LZW

```c
#include <stdio.h>
#include <string.h>

typedef unsigned short int ui16;

#define TAM_DICT 4096
#define NUM_SIMB 256
/*-------------------------------------------------
inicia dicionario
corrente = primeiro simbolo da entrada
while (!fimDaEntrada) {
        simbolo = proximo simbolo da entrada
        if (corrente + simbolo esta no dicionario)
        corrente = corrente + simbolo
        else {
                escreve codigo corrente
                adiciona corrente + simbolo no dicionario
                corrente = simbolo
        }
}
escreve codigo corrente (do ultimo)
-------------------------------------------------*/
void codifica(int *in, int n) {
        ui16 dicionario[TAM_DICT][NUM_SIMB];
        int posDict;          // Posicao no dicionario
        int simbolo;          // Simbolo da entrada
        int corrente = in[0]; // Indice atual da tabela
        // Inicia dicionario
        memset(dicionario, 0, TAM_DICT * NUM_SIMB * sizeof(ui16));
        posDict = NUM_SIMB;
        int i = 1;
        while (i < n) {
                simbolo = in[i++];
                ui16 prox = dicionario[corrente][simbolo];
                if (prox != 0)
                   corrente = prox; // Segue o simbolo no dicionario
                else {
                   printf("[%d] ", corrente);
                   if (posDict < TAM_DICT)
                   dicionario[corrente][simbolo] = posDict++;
                   else { // Reinicia o dicionario
                        memset(dicionario, 0, TAM_DICT * NUM_SIMB * sizeof(ui16));
                        posDict = NUM_SIMB;
                   }
                   corrente = simbolo;
                }
        };
        printf("[%d]\n", corrente);
}

int main() {
        int in[16] =
        {39, 39, 126, 126, 39, 39, 126, 126, 39, 39, 126, 126, 39, 39, 126, 126};
        codifica(in, 16);
        return 0;
}
```

Listagem 2: Implementação da descompressão LZW

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef unsigned short int ui16;

// Esta implementacao usa um esquema de alocacao customizado
// porque o tamanho da sequencia maxima e conhecida.
typedef struct alocaInfo {
        ui16 *base;
        int tam;
        ui16 *proxAloca;
} alocaInfo;

// Inicia a estrutura customizada para alocacao
void iniciaAloca(alocaInfo *aloc, int tam) {
        aloc->base = malloc(tam);
        aloc->tam = tam;
        aloc->proxAloca = aloc->base;
}

// simula o malloc para a alocacao customizada.
ui16 *aloca(alocaInfo *aloc, int len) {
        ui16 *ret = aloc->proxAloca;
        aloc->proxAloca += len;
        return ret;
}

#define TAM_DICT 4096
#define NUM_SIMB 256

/*-------------------------------------------------
inicia dicionario
anterior = primeiro simbolo da entrada
escreve anterior
while (!fimDaEntrada) {
        simbolo = proximo simbolo da entrada
        cadeia = sequencia do anterior no dicionario
        adiciona simbolo + cadeia[0] no dicionario
        escreve sequencia do simbolo no dicionario
        anterior = simbolo
}
-------------------------------------------------*/
void decodifica(ui16 *in, int n) {
        struct {
                ui16 *seq;
                int tam;
        } dicionario[TAM_DICT];
        alocaInfo aInfo;
        ui16 *marca;
        int posDict;
        int anterior; // codigo anterior
        int i;
        // Inicia o dicionario
        iniciaAloca(&aInfo, TAM_DICT * TAM_DICT * sizeof(ui16));
        marca = aInfo.proxAloca;
        for (int i = 0; i < NUM_SIMB; i++) {
                dicionario[i].seq = aloca(&aInfo, 1);
                dicionario[i].seq[0] = i;
```

```
60                  dicionario[i].tam = 1;
61          }
62          posDict = NUM_SIMB;
63
64          anterior = in[0];
65          printf("[%d] ", anterior);
66          i = 1;
67          while (i < n) {
68                  int simbolo = in[i++]; // Proximo simbolo
69                  if (posDict == TAM_DICT)
70                  { // Reinicia o dicionario
71                      aInfo.proxAloca = marca;
72                      for (int i = 0; i < NUM_SIMB; i++) {
73                          dicionario[i].seq = aloca(&aInfo, 1);
74                          dicionario[i].seq[0] = i;
75                          dicionario[i].tam = 1;
76                      }
77                      posDict = NUM_SIMB;
78                  }
79                  else {
80                      // Adiciona a entrada no dicionario.
81                      // A nova entrada e a mesma da entrada anterior mais um simbolo
82                      int tam = dicionario[anterior].tam;
83                      dicionario[posDict].tam = tam + 1;
84                      dicionario[posDict].seq = aloca(&aInfo, tam + 1);
85                      for (int k = 0; k < tam; k++)
86                          dicionario[posDict].seq[k] = dicionario[anterior].seq[k];
87                      if (simbolo == posDict)
88                          dicionario[posDict++].seq[tam] = dicionario[anterior].seq[0];
89                      else
90                          dicionario[posDict++].seq[tam] = dicionario[simbolo].seq[0];
91                  }
92                  // escreve a sequencia do simbolo
93                  for (int k = 0; k < dicionario[simbolo].tam; k++)
94                      printf("[%d] ", dicionario[simbolo].seq[k]);
95                  anterior = simbolo;
96          }
97          free(aInfo.base);
98          puts("");
99  }
100
101 int main() {
102         ui16 in[10] =
103         {39, 39, 126, 126, 256, 258, 260, 259, 257, 126};
104         decodifica(in, 10);
105         return 0;
106 }
```

# Referências


[Gonzalez 2010]GONZALEZ, R. C. *Processamento Digital de Imagens*. [S.l.]: Person Education, 2010. ISBN 978-85-8143-586-2.

[Welch 1984]WELCH, T. A technique for high-performance data compression. *IEEE Computer*, v. 17, n. 6, p. 8–19, 1984.