Trabalho Prático 0
Conversão de Imagens Coloridas Para Tons de Cinza

Douglas Rodrigues Fernandes Filho

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

drff@ufmg.br

1. Introdução

O problema proposto foi implementar um programa para conversão de uma imagem colorida para tons de cinza, na qual cada imagem convertida seria uma amostra de um espaço de cores retratadas em tons de cinza.

Para a conversão de um pixel colorido em seu nível aproximado de cinza existem diversas estratégias diferentes e a escolhida para esse trabalho foi a seguinte estratégia:

$$Y = \frac{49}{255}(0.30R + 0.59G + 0.11B)$$

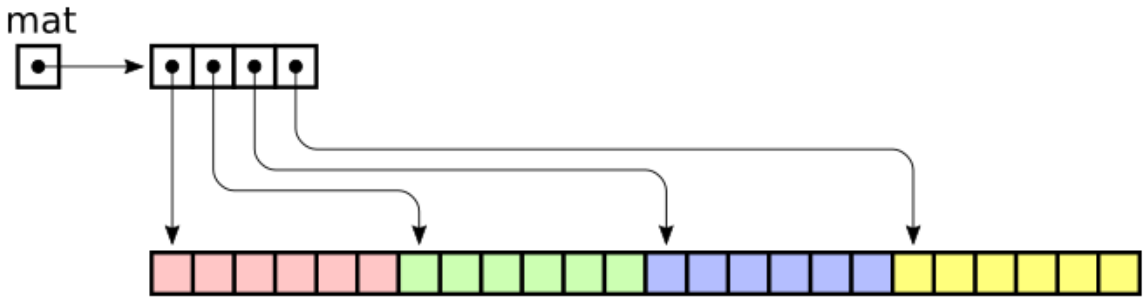
Sendo R, G e B as cores primitivas vermelho, verde e azul do pixel original e Y correspondente ao tom de cinza mapeado.

2. Implementação

O Programa foi desenvolvido na linguagem C++, compilado pelo compilador G++ versão 11.2.0 (Ubunto).

2.1 Estrutura de Dados

A implementação do programa teve como base a estrutura de dados de uma matriz com a alocação de um vetor de ponteiros de linhas contíguas. Nesse tipo de matriz temos um vetor de ponteiros para linhas, como se vê normalmente, porém as linhas são alocadas como um único vetor de elementos.



Essa implementação foi escolhida pela fato de que quando todos os elementos da matriz entrando alocados sequencialmente na memória temos um melhor aproveitamento da localidade de referência espacial, dado que os acessos de memória das funções implementadas são sempre feitos de forma sequencial sempre se sem uma ótima proximidade entre os elementos que serão buscados na memória.

2.2 Classes

Foi utilizada apenas uma classe no projeto, sendo ela a abstração das matrizes que foram manipuladas durante todo o programa. A classe **mat.h** consiste em uma simples implementação de uma matriz dinamicamente alocada e possui a seguinte estrutura:

```
#include "memlog.h"
#include <fstream>

typedef struct mat{
    int **m;
    int linhas, colunas;
    int id;
} mat_tipo;

void criaMatriz(mat_tipo * mat, int tx, int ty, int id);
void inicializaMatrizPPM(mat_tipo * mat, char *arq);
void inicializaMatrizPGM(mat_tipo * matPGM, mat_tipo * matPPM);
void salvaImagemPGM(mat_tipo * mat);
```

A função **criaMatriz** recebe um endereço de memória, a dimensões e o id referente a matriz que se deseja criar e faz a alocação de memória da mesma.

A função **inicializaMatrizPPM** recebe o endereço de memória de uma matriz e o arquivo PPM, a partir disso ela constrói na matriz uma réplica dos valores do arquivo para que posteriormente esses valores possam ser utilizados sem a necessidade de acessar o arquivo novamente.

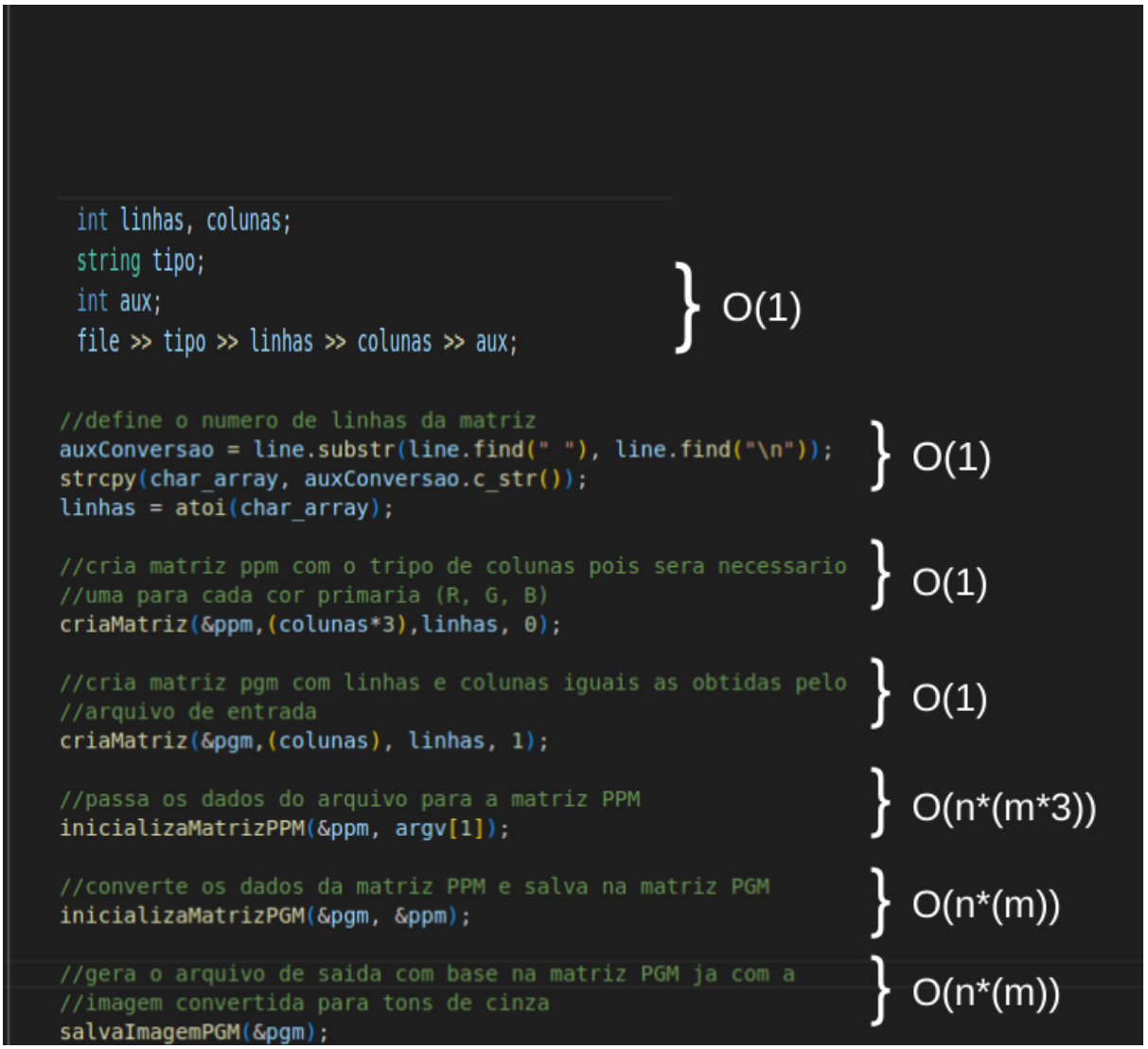
A função **inicializa Matriz PGM** recebe o endereço de memória de uma matriz PGM e uma matriz PPM e faz a conversão dos valores existentes na matriz PPM para a matriz PPM, de acordo com a função mostrada acima.

A função **salvaImagemPGM** recebe uma matriz PGM e a partir dela faz a montagem da imagem que será salva pelo programa, já convertida em tons de cinza.

3. Análise de complexidade

3.1 Tempo

Para a análise de tempo iremos considerar a criação e destruição das matrizes, assim como a leitura de uma linha em separado do arquivo, como **O(1)**.



Inicializar a Matriz PPM consiste em passar em cada uma das posições dela populando com os valores necessários, portanto temos um **for()** de 0 a n(linhas) e um **for()** dentro dele ate m*3(colunas), pois como explicitado anteriormente a matriz PPM possui três vezes mais colunas que a matriz PGM.

Tanto inicializar a matriz PGM quanto salvar a imagem PGM manipulam a matriz PGM que é percorrida com a mesma estrutura explicada acima, porém com **n** linhas e **m** colunas.

Portanto temos a complexidade dada por:

$$O(1) + O(1) + O(1) + O(1) + O(n*m) + O(n*m) + O(n*(m*3)) = O(n*m)$$

3.2 Espaço

Para a análise de espaço iremos considerar a alocação dinâmica das matrizes PPM e PGM. No programa temos a necessidade de criar uma estrutura para cada uma das matrizes nos gerando assim a:

$$O(n*m) + O(n*(m*3)) = O(n*m)$$

4. Robustez

Para a robustez do programa foi utilizada a biblioteca msgassert.h, como solicitado no enunciado do trabalho. A biblioteca foi utilizada em conjunto com as ferramentas já disponíveis em c, como o try catch, para garantir uma maior robustez ao programa.

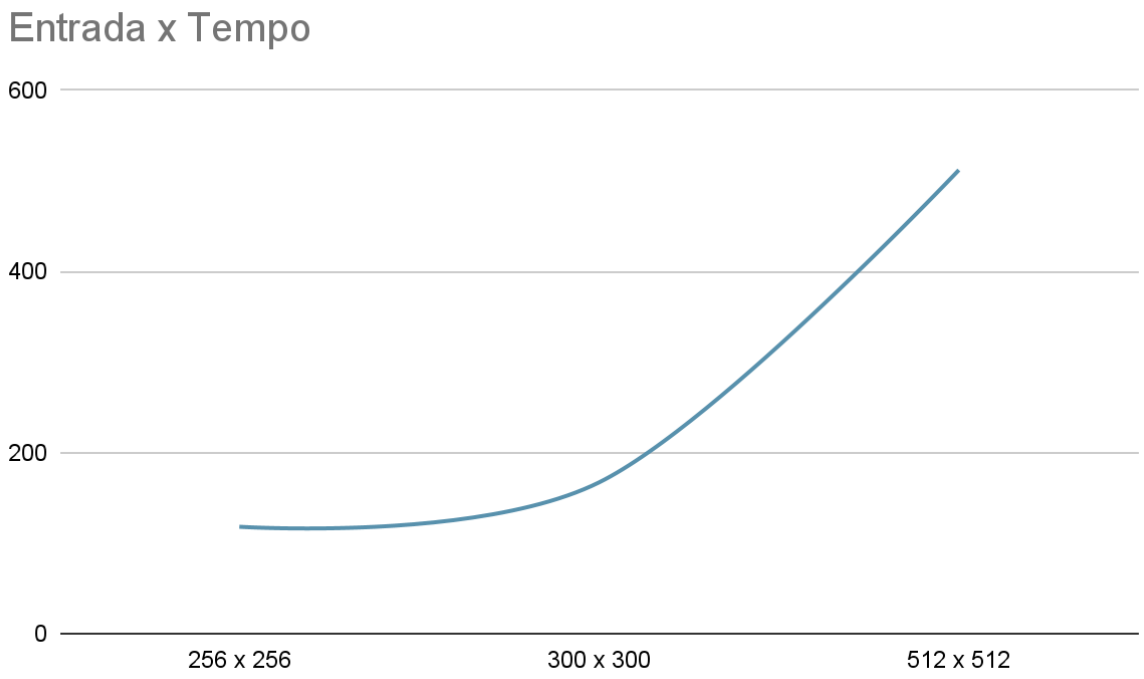
```
ifstream file(nomeImagemPPM);

if(file.is_open()) {⋯
}else{
    erroAssert(file.is_open(),"Arquivo não encontrado");
}
```

5. Análise Experimental

O gráfico abaixo representa o crescimento que pode ser observado na complexidade dada pela análise de complexidade feita anteriormente. Isso fica bem claro se compararmos o tempo de execução de uma entrada 256 x 256 e de uma entrada 512 x 512, o crescimento no gráfico é claramente não linear, não seguindo o dobro de tempo para o dobro de entrada.

Intermediariamente ainda podemos observar uma imagem 300x300 que não apresenta um crescimento tão elevado como a de 512x521, o que é uma característica de uma curva de crescimento exponencial.



6. Conclusões

O trabalho foi interessante para entender a dinâmica de como tipos abstratos de dados podem representar coisas mais palpáveis como imagens, e também entender a importância de como manipular os dados de forma correta e consistente é importante para não perder nenhum tipo de informação.

As maiores dificuldades enfrentadas foram relacionadas às bibliotecas disponibilizadas pelos professores para fazer as análises, principalmente a geração dos gráficos.

7. Compilação

Para a compilação basta rodar o comando **make** e posteriormente rodar **./bin/tp0 -i “imgaem_entrada” -o “imagem_saida” -p log.out -l** para rodar o arquivo compilado com a entrada desejada.

8. References

Alocação dinâmica de matrizes. *https://www.inf.ufpr.br/roberto/ci067/14_alocmat.html*
Slides Estrutura de Dados.