Trabalho Prático 0

Conversão de imagem colorida para tons de cinza

Matheus Guimarães Couto de Melo Afonso
Matrícula: 2021039450

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

matheusgcma@ufmg.br

1. Introdução

O trabalho consistiu em implementar, utilizando estruturas de dados, um programa que recebe pela linha de comando um arquivo do tipo .ppm (Portable Pixmap Format), que contém informações de uma imagem colorida – em escala RGB –, e convertê-lo para o formato .pgm (Portable Graymap Format), que armazena uma imagem em escala de cinza. A conversão é feita por meio da seguinte fórmula:

$$Y = \frac{49}{255}(0.3R + 0.59G + 0.11B)$$

onde R, G e B, são respectivamente os tons de vermelho, verde e azul da imagem original, e Y é o tom de cinza correspondente.

A seção 2 contém uma explicação mais profunda sobre as estruturas de dados e métodos implementados, bem como informações sobre o ambiente de desenvolvimento. A seção 3 apresenta uma análise de complexidade do código, enquanto a seção 4 se aprofunda nas estratégias de robustez aplicadas no desenvolvimento. Na seção 5, mostro os resultados dos testes de desempenho do programa, e por fim a seção 6 conclui a documentação do trabalho.

2. Método

2.1. Organização do projeto

O projeto é estruturado no modelo sugerido pelos professores: a pasta **src** armazena os arquivos de código (*.c e *.cpp) e a pasta **include** têm os arquivos de cabeçalho (*.h). Ao compilar o código por meio do **Makefile**, que está na pasta raiz do projeto, os arquivos *.o são gerados no diretório **out** e o arquivo executável **run.out** é gerado na pasta **bin**. A única diferença na organização é a existência da pasta **assets**, onde são colocados alguns arquivos *.ppm de exemplo no diretório **assets/in**, e o arquivo

.pgm de saída é gerado em **assets/out/output.pgm** caso o usuário não informe o local de saída desejado.

2.2. Estruturas de dados

Foram implementadas duas estruturas de dados para armazenamento das informações das imagens *.ppm e *.pgm: as *structs* PpmImg e PgmImg, que tem uma estrutura bem parecida semelhante.

```cpp
struct PpmImg {
    std::string path;
    int width;
    int height;
    int **red;
    int **green;
    int **blue;

    PpmImg() {}
    PpmImg(std::string p) : path(p) {}
};
```

```cpp
struct PgmImg {
    std::string path;
    int width;
    int height;
    int **gray;

    PgmImg() {}
    PgmImg(std::string p) : path(p) {}
};
```

A propriedade *path* armazena o caminho das imagens, enquanto as dimensões são salvas em *width* e *height*. A diferença entre as estruturas é que a PpmImg tem três matrizes para salvar os dados da imagem, já que uma imagem .ppm tem informações sobre tons de vermelho, verde e azul, enquanto a imagem .pgm tem apenas uma matriz para salvar os dados dos tons de cinza após a conversão.

O programa também implementa a *struct* Flags, que serve apenas para salvar os diferentes argumentos de linha de comando que o programa pode receber, como o caminho para arquivos de entrada e saída ou se devem ser salvos os *logs* de registro de desempenho.

```cpp
struct Flags {
    std::string input_file;
    std::string output_file;
    bool save_logs;
    std::string log_file;
    bool save_mem_acc;

    Flags()
        : input_file("assets/in/mineirao.ppm"),
          output_file("assets/out/output.pgm"),
          save_logs(false),
          save_mem_acc(false) {}
};
```

2.3. Funções

O programa implementa três funções principais: *readPpmImg*, *convertPpmToPgm* e *writePgmImg*.

- readPpmImg: recebe como parâmetro um objeto PpmImg, aloca dinamicamente as matrizes dele e percorre o arquivo de entrada salvo na

propriedade *path* do objeto PpmImg, salvando os valores de cada pixel no objeto.

- convertPpmToPgm: recebe como parâmetro um objeto PpmImg e um PgmImg, aloca dinamicamente a matriz *gray* do PgmImg e realiza a conversão do valor de cada pixel utilizando uma função auxiliar, posteriormente salvando o resultado dessa operação no endereço correspondente da matriz.
- writePgmImg: recebe uma PgmImg como parâmetro e escreve a matriz *gray* no arquivo de saída salvo na propriedade *path* do objeto PgmImg.

2.4. Ambiente de desenvolvimento

- Sistema operacional: WSL Ubuntu 20.4
- Linguagem de programação: C++
- Compilador: g++ Ubuntu 9.4.0-1ubuntu1
- Processador: Intel Core i5-9400F @2.90GHz
- Memória RAM: 2 x 8GB

3. Análise de Complexidade

A complexidade das funções do programa depende do tamanho da entrada. Como nesse programa a entrada é uma imagem, vamos utilizar a notação m para representar a **altura** da imagem e n para representar a **largura**. A seção está dividida em duas partes – complexidade de tempo e espaço – e em cada uma delas analiso as três funções individualmente.

3.1. Complexidade de tempo

- readPpmImg: Aloca dinamicamente três matrizes $m \times n$ em um laço ($3O(m)$) e depois percorre o arquivo *.ppm, escrevendo cada pixel nas posições correspondentes das matrizes *red, green* e *blue* ($O(mn)$, uma vez que acontece em dois laços aninhados). A complexidade total é dada então por:

$$3O(m) + O(mn) = O(max(m, mn)) = O(mn)$$

- convertPpmToPgm: Aloca dinamicamente uma matriz $m \times n$ em um laço ($O(m)$) e depois percorre as matrizes *red, green* e *blue* do objeto ppm, chamando um método auxiliar que realiza as conversões de RGB para escala de cinza e salvando o retorno na posição correspondente da matriz *gray* ($O(mn)$, uma vez que acontece em dois laços aninhados). A complexidade total é dada por:

$$O(m) + O(mn) = O(max(m, mn)) = O(mn)$$

- writePgmImg: Escreve o cabeçalho do arquivo .pgm de saída ($3O(1)$) e depois itera na matriz *gray* do objeto PgmImg, escrevendo o valor de cada pixel no arquivo ($O(mn)$). A complexidade total é dada por:

$$3O(1) + O(mn) = O(max(1, mn)) = O(mn)$$

3.2. Complexidade de espaço

- readPpmImg: A alocação de três matrizes $m \times n$ tem complexidade de espaço $3O(mn)$. O armazenamento da altura e largura da imagem tem complexidade constante $2O(1)$. A complexidade de espaço total portanto é:

$$3O(mn) + 2O(1) = O(max(mn, 1)) = O(mn)$$

- convertPpmToPgm: De forma semelhante à função readPpmImg, aqui há a alocação de uma matriz $m \times n$ ($O(mn)$) e o armazenamento da altura e largura da imagem ($2(O(1))$. Complexidade de espaço total:

$$O(mn) + 2O(1) = O(max(mn, 1)) = O(mn)$$

- writePgmImg: A função writePgmImg não escreve nada na memória, portanto tem complexidade de espaço igual a $O(0)$.

4. Estratégias de Robustez

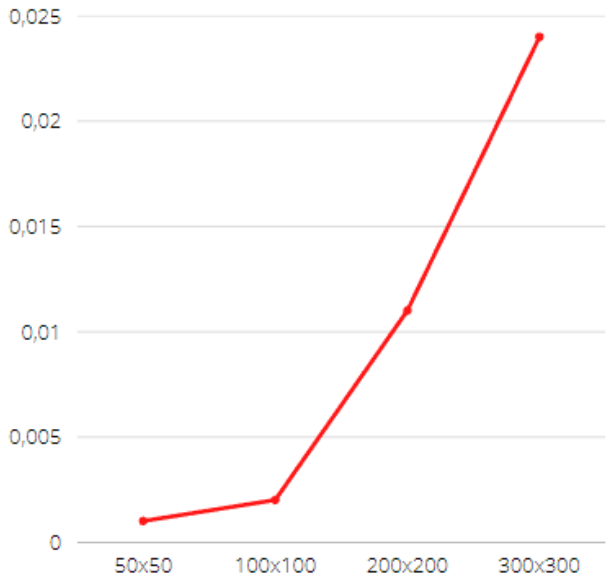
A robustez de um código diz respeito à sua capacidade de funcionar corretamente mesmo em condições anormais, por meio de mecanismos de programação defensiva e tolerância a falhas. A robustez do programa foi implementada por meio da biblioteca **msgassert.h**, disponibilizada pelos professores de Estrutura de Dados.

Após receber e interpretar os argumentos recebidos na linha de comando, é utilizada a função *avisoAssert*, para informar o usuário onde estão o arquivo de entrada, saída e os logs de execução padrões, que são utilizados no caso de o usuário não passar estas informações como parâmetro.

Mais a frente, na função *readPpmImg*, há uma verificação do tamanho da imagem .ppm de entrada. Caso a altura ou largura sejam negativas, a execução do programa é abortada. A função *readPpmImg* também verifica o formato do arquivo de entrada: quando a entrada não é .ppm, a execução é abortada. Por último, a função *writePgmImg* aborta a execução quando o formato do arquivo de saída não é .pgm.

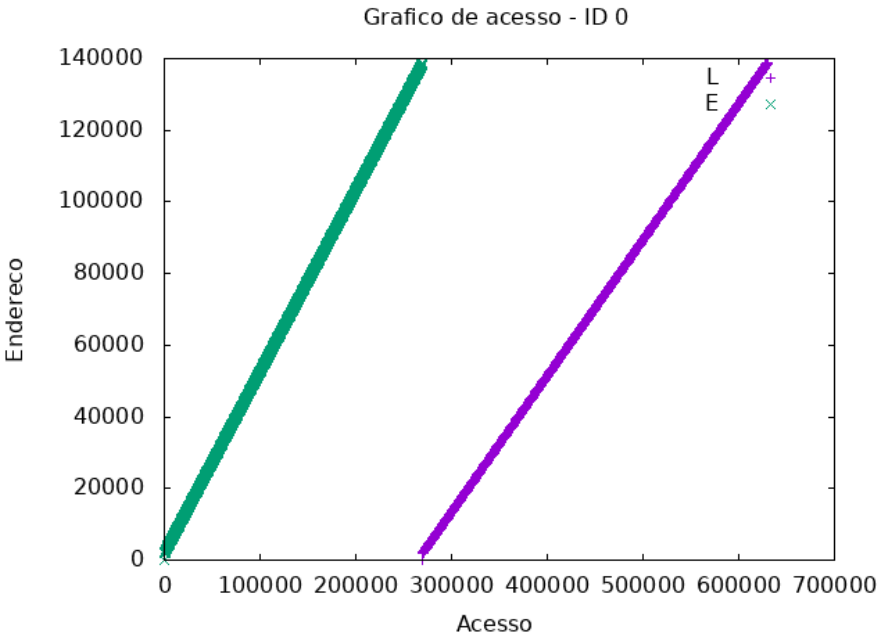
5. Análise Experimental

Na seção de análise experimental, são apresentados os resultados de testes feitos em cima do programa, que verificam fatores como tempo de execução, localidade de referência e distância de pilha. São utilizadas como ferramentas da análise as bibliotecas **memlog** e **analisamem**, disponibilizadas pelos professores da disciplina.

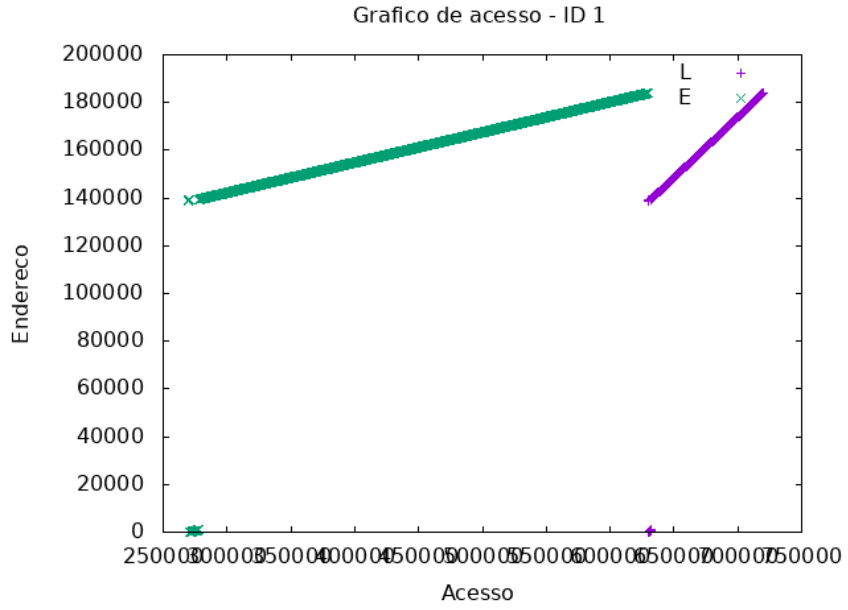


O gráfico acima ilustra o tempo de execução do programa conforme as dimensões da imagem .ppm de entrada aumentam. A curva que resulta da análise desses tempos de execução representa uma função quadrática, algo previsto pela análise de complexidade.

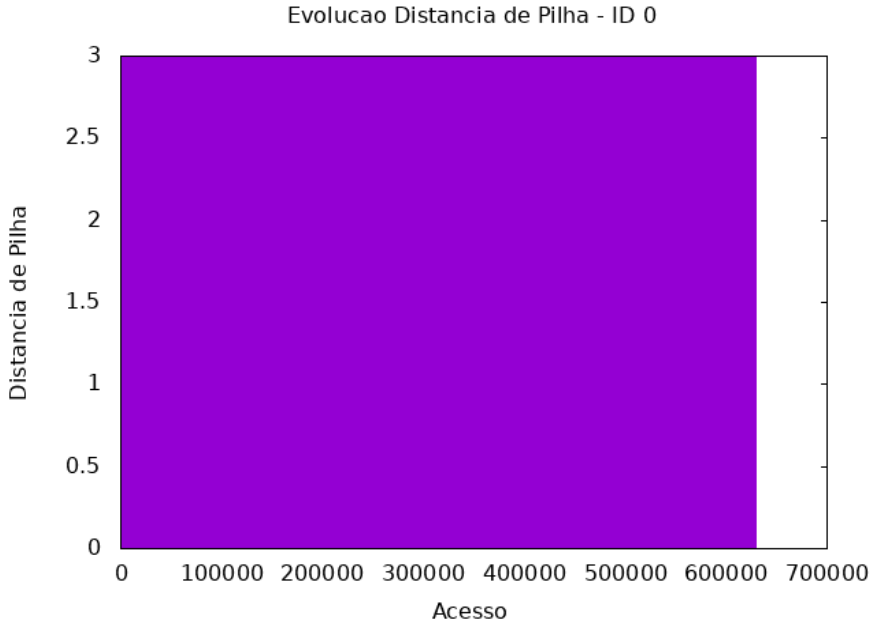
5.1. Análise de padrão de acesso a memória e localidade de referência

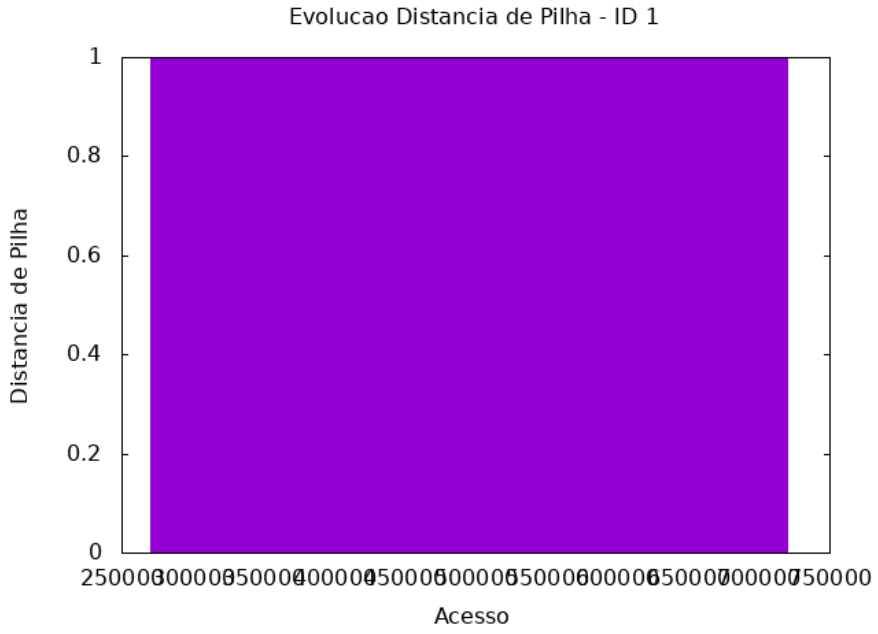


O gráfico acima mostra os registros de leitura e escrita na estrutura de dados PpmImg. É possível perceber que o gráfico é consistente com a realidade, uma vez que as escritas na estrutura começam bem no início da execução do programa, na função *readPpmImg*, enquanto as operações de leitura ocorrem mais à frente na função *convertPpmToPgm*.



Já o segundo gráfico, da estrutura de dados PgmImg, tem um desenho um pouco diferente: para cada escrita, são realizadas 3 leituras na estrutura PpmImg, então as escritas aqui (ocorrem na função *convertPpmToPgm*) são mais distribuídas ao longo do tempo de execução. As operações de leitura, por outro lado, se concentram num espaço de tempo menor, uma vez que não dividem tempo de execução com mais nada na função *writePgmImg*.





Os gráficos de distância de pilha das duas estruturas são bem parecidos, são realizados muitos acessos com distâncias de pilha constantes, sendo 3 para PpmImg e 1 para PgmImg. Esses números não aparecem ao acaso: na PpmImg a distância é 3 porque ela opera com três matrizes, enquanto a PgmImg opera em apenas uma matriz.

6. Conclusões

O trabalho prático 0 tratou da implementação de um programa em C++ que converte uma imagem colorida em formato .ppm para uma imagem em escala de cinza no formato .pgm. As imagens podem ter um tamanho arbitrariamente grande, uma vez que as matrizes nas quais são armazenadas as imagens são alocadas dinamicamente.

No trabalho tive um aprendizado muito grande nas questões de análise de complexidade e de localidade de referência, conceitos muito importantes na área de desenvolvimento de software e com os quais eu nunca havia tido contato antes. Além disso, o trabalho ajudou a refrescar e praticar muitas coisas da linguagem C++ que eu não utilizava há muito tempo, como a alocação dinâmica de memória e a leitura e escrita de arquivos.

7. Bibliografia

Slides e códigos da disciplina Estrutura de Dados. Disponibilizados via Moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

How to get file extension from string in C? Disponível em: https://stackoverflow.com/questions/51949/how-to-get-file-extension-from-string-in-c Acesso em 18 set. 2022.

8. Instruções para compilação e execução

- Acesse o diretório /tp do projeto por meio de um terminal
- Digite o comando **make** para compilar o programa e gerar o arquivo executável **run.out** na pasta /bin
- Ao executar o comando **bin/run.out**, o programa é executado com a entrada (assets/in/mineirao.ppm), saída (assets/out/output.pgm) e arquivo para registro de logs (bin/log.out) padrões.
- Para utilizar arquivos além dos padrões, o usuário pode inserir algumas opções na linha de comando:
    - -i <arquivo_de_entrada> para mudar a entrada .ppm
    - -o <arquivo_de_saida> para mudar o local da saída .pgm
    - -p <arquivo_de_logs> para mudar onde serão escrito os logs
    - -l para registrar ou não os acessos à memória nos logs