UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Universidade Federal de Minas Gerais

Turma: Ciência da Computação **Prof.:** Gisele e Wagner

Nome: Rubia Alice Moreira de Souza
Matrícula: 2022043507

TP0 - Conversor de Imagens

Belo Horizonte
2022

Sumário

1. Introdução

Este trabalho é voltado à manipulação de arquivos de imagem nos formatos .PPM e .PGM. O seu principal objetivo é criar um conversor de arquivos .PPM que possuem imagens com cores para uma imagem em tons de cinza no formato .PGM. A conversão foi implementada ao aplicar a fórmula abaixo em cada pixel RGB da imagem .PPM resultando em um tom de cinza que formará um pixel da nova imagem.

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

Além disso, foi criado um leitor de arquivo .PPM para conseguir extrair os pixels que formam a sua imagem e um escritor de arquivo .PGM para gerar a nova imagem.

2. Método

2.1. Classes

O programa foi organizado em diversas classes. Inicialmente, temos as classes voltadas a representar as imagens que iremos manipular:

Pixel	
Descrição: É uma representação genérica de um pixel presente em uma imagem qualquer.	
Propriedades	**Métodos**
tomMaximo : short	Pixel() Pixel(tomMaximo : short) setTomMaximo(tomMaximo : short) : void getTomMaximo() : void

PPMPixel : Pixel	
Descrição: É a representação de um pixel que possui a combinação das cores RGB.	
Propriedades	**Métodos**
vermelho : short verde : short azul : short	setVermelho(vermelho : short) : void getVermelho() : short void setVerde(verde : short) : void getVerde() : short void setAzul(azul : short) getAzul() : short

PGMPixel : Pixel	
Descrição: É a representação de um pixel em tons de cinza.	
Propriedades	**Métodos**

valor : short	setValor(valor : short) : void getValor() : short

Imagem	
Descrição: É a representação de uma imagem em um formato qualquer que possui uma lista de pixels genéricos (podendo ser PGM ou PPM)	
Propriedades	**Métodos**
tipo : string altura : unsigned int largura : unsigned int quatidadeTons : short pixels : Lista<Pixel*>*	setTipo(tipo : string) : void getTipo() : string setAltura(altura : unsigned int) : void getAltura() : unsigned int setLargura(largura : unsigned int) : void getLargura() : unsigned int setQuantidadeTons(quatidadeTons : short) : void getQuantidadeTons() : short getPixels() : Lista<Pixel*>* getTotalPixels() : unsigned int

Além disso, temos as classes que realizam a manipulação de arquivos:

ArquivoHandler	
Descrição: É uma classe abstrata com o objetivo de unificar as classes que lidam com a manipulação de arquivos, independentemente do seu tipo.	
Propriedades	**Métodos**
caminho : string	ArquivoHandler(caminho : string) void setCaminho(camiho : string) getCaminho() : string

PPMReader : ArquivoHandler	
Descrição: Tem a funcionalidade de ler um arquivo de imagem no formato .PPM e armazenar suas informações em uma classe “Imagem”.	
Propriedades	**Métodos**
	PPMReader(caminho : string) lerImagem() : Imagem*

PGMWriter : ArquivoHandler

Descrição: Tem a função de criar um arquivo de imagem no formato .PGM a partir de uma classe “Imagem”.	
Propriedades	**Métodos**
	PGMWriter(caminho : string) criarImagem(imagem : Imagem) : void

ConversorImagem	
Descrição: Responsável por converter as imagens de PPM para PGM pixel a pixel.	
Propriedades	**Métodos**
	converterPPMparaPGM(imagemPPM : Imagem) : Imagem*

Observação: O ConversorImagem pode ser convertido posteriormente para uma interface em que diferentes implementações irão herdar dela de modo a seguir o princípio Open/Closed e Dependency Inversion do SOLID.

Por fim, temos uma estrutura de lista duplamente encadeada de tipo genérico para que ela possa armazenar os Pixels presentes na classe de imagem.

Lista<T>	
Descrição: É uma estrutura de dado em formato de lista duplamente encadeada.	
Propriedades	**Métodos**
tamanho : unsigned int inicio : Celula<T>* fim : Celula<T>*	adicionarFim(valor : T) : void adicionarComeco(valor : T) : void adicionarNaPosicao(valor : T, posicao : unsigned int) : void removerFim() : void removerComeco() : void remover(posicao : unsigned int) : void limpar() : void getTamanho() : unsigned int get(posicao : unsigned int) : T getInicio() : Celula<T>* getFim() : Celula<T>* estaVazia() : bool

Além disso, temos exceções customizadas para verificar possíveis erros que possam ocorrer na leitura, escrita das imagens, na manipulação da lista e na própria execução da main. Elas serão abordadas no tópico 4. Estratégias de Robustez.

2.2. Fluxo de Execução

O fluxo de execução se inicia no arquivo main.cpp. Nele, é chamada a classe PPMReader para realizar a leitura do arquivo fornecido nos parâmetros de entrada do executável. Após isso, o ConversorImagem é utilizado para alterar os pixels para tons de cinza. Por fim, a imagem resultante é gravada utilizando a classe PGMWriter.

2.3. Configurações de Ambiente

O código foi executado nas seguintes configurações de ambiente:

- **Sistemas operacionais:** Windows, Linux e na WSL do Windows utilizando a versão Ubuntu do Linux
- **Linguagem utilizada:** C++11
- **Compilador utilizado:** g++ da GNU Compiler Collection
- **Processador utilizado:** Intel(R) Core(™) i5-7200U CPU @ 2.50GHz, 2701 Mhz, 2 Núcleo(s), 4 Processador(es) Lógico(s)
- **Quantidade de memória RAM disponível:** 8Gb

3. Análise de Complexidade

Neste contexto, estamos lidando com uma matriz como entrada que representa a imagem. O atributo variável dessa matriz são suas dimensões. Podemos assumir que “L” representa o número de pixels em uma linha e “C” representa o número de pixels em uma coluna. Então, como nossa entrada de dados variáveis no programa é a quantidade de pixels, nós podemos expressá-la como “L x C” (Quantidade de pixels nas linhas vezes a quantidade deles nas colunas). Portanto, podemos dizer que nosso “n = L x C”, sendo “n” a entrada das funções. A partir disso, temos três funções principais que irão variar segundo o tamanho da entrada “n”. Nós temos a função de leitura, conversão e escrita.

Primeiramente, temos a função de leitura, “lerImagem()”. Ela tem a responsabilidade de ler um arquivo .PPM e gerar uma instância da classe Imagem a partir dos seus dados. Para isso, ela armazena os atributos presentes nas primeiras linhas do arquivo em seus respectivos campos na classe de imagem e concatena todos os pixels em uma lista. Essa função sempre irá percorrer todos os pixels da imagem a fim de armazená-los e o laço para isso sempre irá depender do tamanho “n” da entrada. Então, temos que a ordem dessa função é O(n). Ao ser executada, ela deve alocar uma quantidade “n” de Pixels em sua lista. Assim, a sua função de complexidade de espaço possui uma ordem de tamanho O(n). Por fim, a função apenas utiliza variáveis auxiliares constantes, ou seja, da ordem de Θ(1).

lerImagem()		
Tipo Função de Complexidade	**Função**	**Ordem**
Tempo	2n + 28	O(n)
Espaço	n + 6	O(n)
Espaço Auxiliar	6	Θ(1)

Após isso, a função de conversão de pixels, “converterPPMparaPGM()”, é executada. Nela, percorremos o vetor de pixels da Imagem lida anteriormente, que possui tamanho “n”. Assim, novamente, essa função depende de “n” e, como precisamos percorrer cada pixel apenas uma vez, ela é da ordem de O(n). Além disso, é necessário criar “n” variáveis auxiliares do tipo Pixel para armazenar a conversão. Assim, o espaço auxiliar é da ordem de Θ(n). Somado a variável de tamanho “n” recebida por parâmetro, teríamos que a função de espaço contém 2n, sendo assim da ordem de O(n).

converterPPMparaPGM()		
Tipo Função de Complexidade	**Função**	**Ordem**
Tempo	5n + 7	O(n)
Espaço	2n + 12	O(n)
Espaço Auxiliar	n + 12	Θ(n)

Por fim, realizamos a escrita da lista de pixels em um arquivo .PGM utilizando a função “criarImagem()”. Nela, também é necessário percorrer sempre toda lista de pixel a fim de poder gravá-los no arquivo que estamos gerando. Como cada elemento da lista é percorrido uma única vez e ela possui tamanho “n”, temos que esta função possui sua função de complexidade na ordem O(n). Como ela recebe apenas uma imagem que será percorrida, a sua função de complexidade de espaço depende das dimensões “n” da imagem, tendo a ordem de O(n). Todas as variáveis auxiliares nessa função são constantes Θ(1).

criarImagem()		
Tipo Função de Complexidade	**Função**	**Ordem**
Tempo	4n + 11	O(n)
Espaço	n + 5	O(n)
Espaço Auxiliar	5	Θ(1)

Portanto, ao avaliarmos a main, com base nas funções de complexidade anteriores, temos que a complexidade do programa, considerando a manipulação das imagens como operação mais importante, seria:

main()		
Tipo Função de Complexidade	**Função**	**Ordem**
Tempo	3n + 6	O(n)
Espaço	2n + 6	O(n)
Espaço Auxiliar	6	Θ(1)

Observação: Instruções relacionadas ao “memlog.h” foram desconsideradas, pois em uma ambiente de desenvolvimento, essas funções não devem ser propagadas para a versão final do programa, servindo apenas como debug e avaliação. Portanto, elas não devem afetar a complexidade da versão final distribuída.

4. Estratégias de Robustez

Para verificar possíveis erros durante a execução foram utilizadas exceções customizadas que herdam de runtime_error e também a biblioteca “msgassert.h”. Esses erros são verificados nas seguintes classes:

L**ista:** Ocorre verificações a fim de validar o index ao adicionar, remover ou buscar um item a fim de evitar posições inválidas não alocadas. Para esses casos, a exceção IndexInvalidoException é lançada.

PGMWriter: É realizada a verificação se foi possível criar o arquivo e abrir caminho de saída. Caso contrário, é lançada a exceção CriacaoArquivoSaidaException.

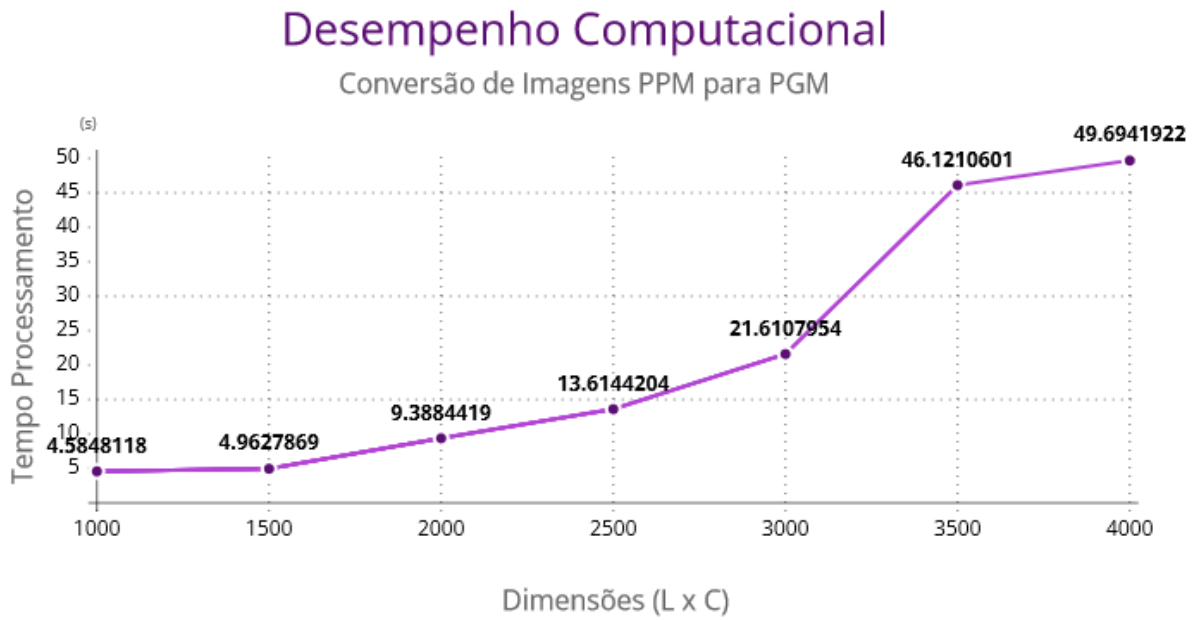
PPMReader: São verificadas 3 exceções

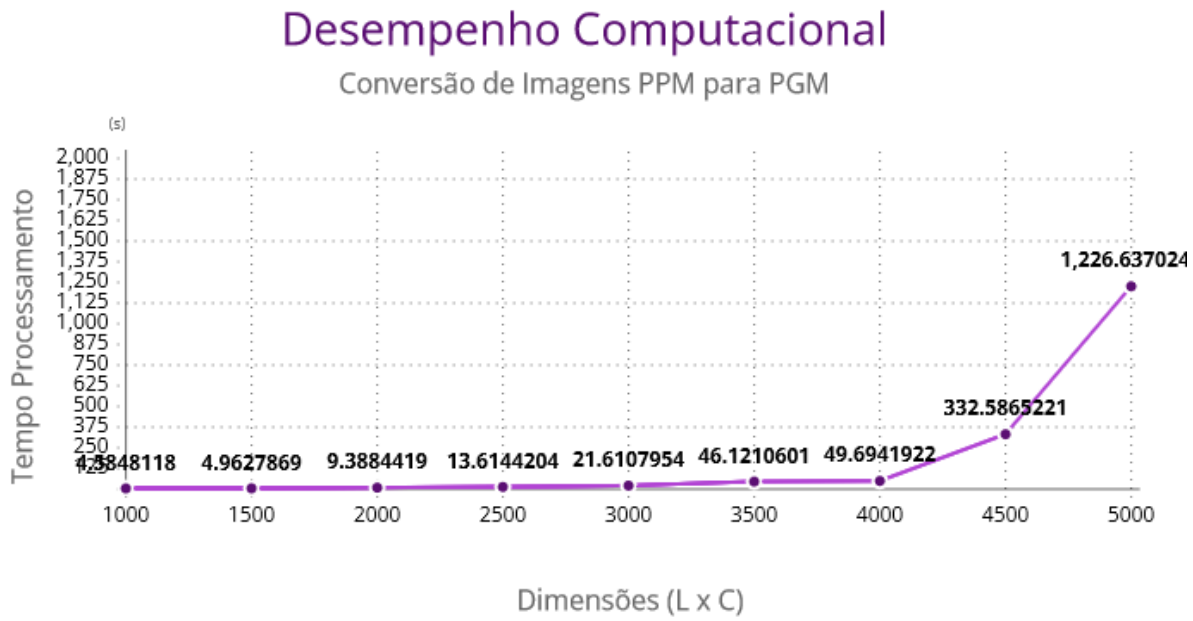
1. Verifica se foi possível abrir arquivo de leitura. Caso contrário, é lançada a exceção: FalhaAbrirArquivoException.
2. Verifica se houve perda de informação durante a leitura do arquivo. Caso tenha ocorrido, é lançada a exceção: LeituraException.
3. Verifica se o formato ASCII do arquivo de entrada é P2. Caso contrário, é lançada a exceção: FormatoInvalidoException.

Além disso, também é verificado se o nome do arquivo para a flag “-p” foi informado. Caso não tenha sido, ele é nomeado por padrão como “log.out” e posicionado no mesmo local que o executável.

5. Análise Experimental

5.1. Desempenho Computacional





Como dito na análise assintótica, nossa função de leitura, conversão e escrita depende diretamente das dimensões da matriz, sendo a entrada “n = L x C” (Quantidade de linhas vezes colunas). Portanto, neste caso em que ambas são iguais, conseguimos ver que a linha do gráfico expressa comportamento esperado de crescimento exponencial à medida que as entradas aumentam. Isso, pois “n = L x C”, como L = x e C = x, logo $n = x^2$, sendo x a dimensão da matriz.

5.2. Análise de Memória

5.2.1. Gráfico de Acesso

5.2.1.1. Imagem PPM

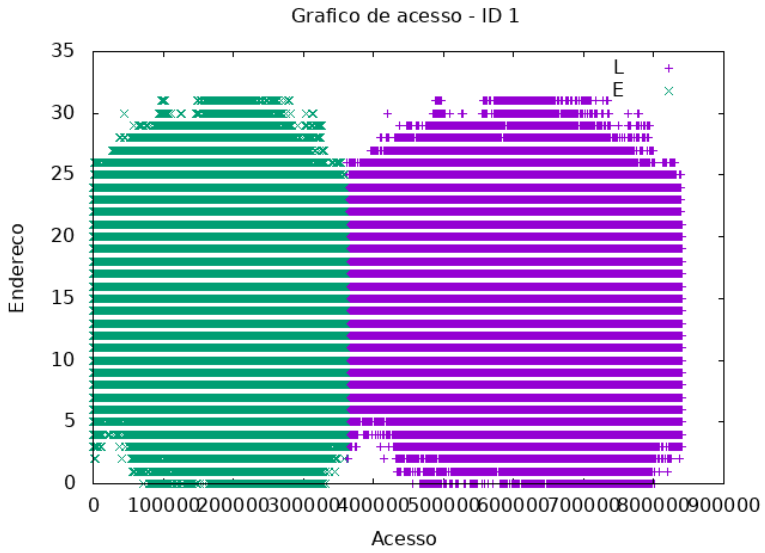


Gráfico de Acesso dos atributos dos Pixels na Lista da Imagem PPM

5.2.1.2. Imagem PGM

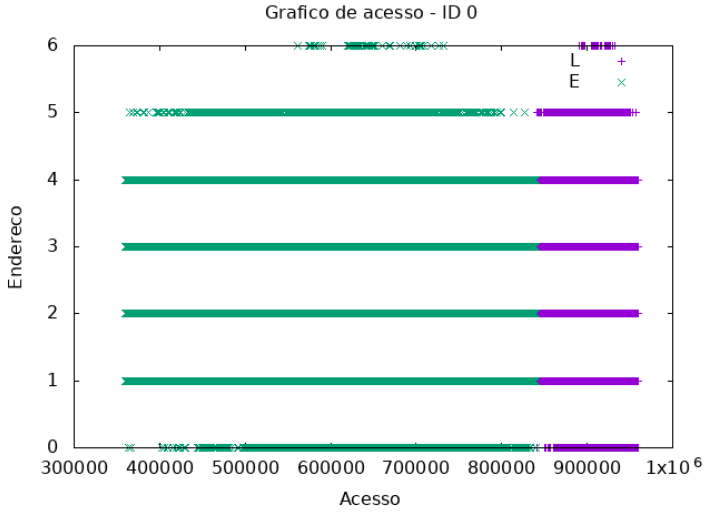
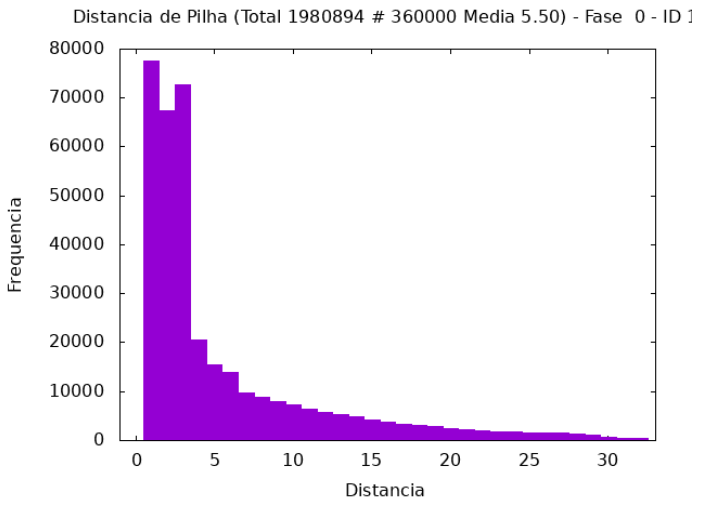


Gráfico de Acesso dos atributos dos Pixels na Lista da Imagem PGM

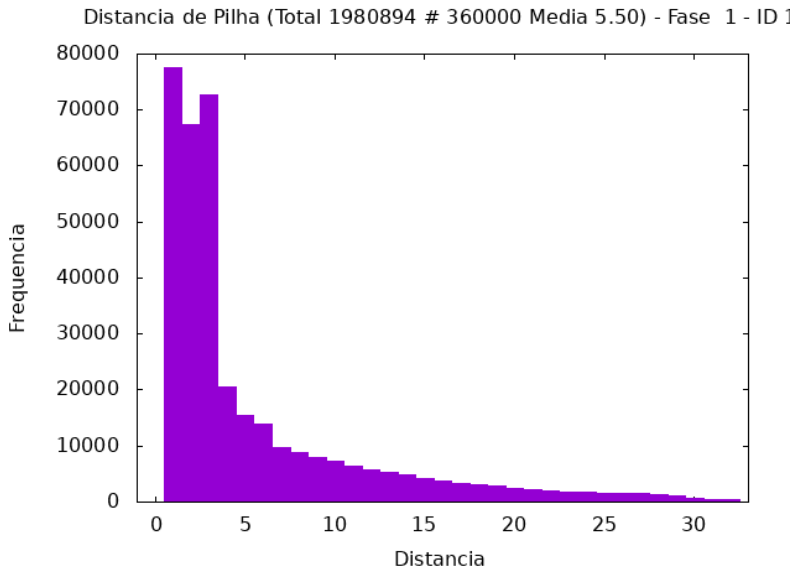
5.2.2. Distância de Pilha

5.2.2.1. Leitura da Imagem PPM

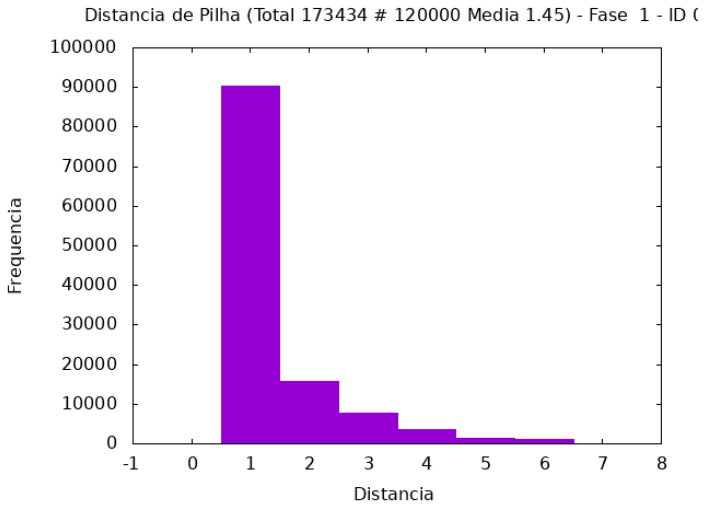


Distância de Pilha durante a criação da lista de Pixels da imagem PGM

5.2.2.2. Conversão de PPM para PGM

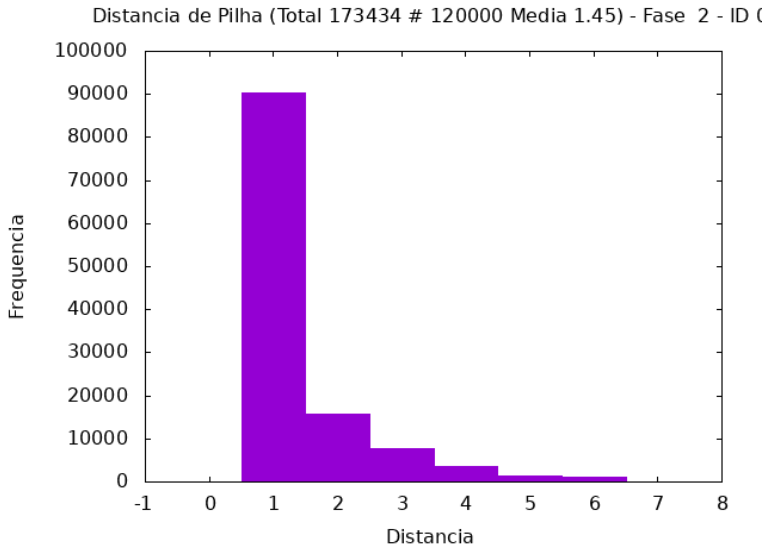


Distância de Pilha durante a leitura da lista de Pixels da imagem PPM



Distância de Pilha durante a leitura da lista de Pixels da imagem PGM

5.2.2.3. Escrita da Imagem PGM



Distância de Pilha durante a escrita da imagem PGM no arquivo de saída

6. Conclusões

Em suma, neste trabalho realizamos a criação de um conversor de imagens coloridas para tons de cinza. A principal dificuldade nesse processo foi conseguir compreender qual é a estrutura e organização dos arquivos .PPM e .PGM que foram manipulados. No entanto, a solução prática foi simples, sendo basicamente criar um leitor e escritor de arquivos, semelhante aos arquivos de texto padrões. Durante o trabalho, foi possível revisar os conceitos de orientação a objetos e boas práticas, como o SOLID, de C++, estudar as “bibliotecas” “memlog.h” e “msgassert.h” e conhecer os formatos de imagem .PGM e .PPM.

7. Bibliografia

Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados.

Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Apêndice 1. Instruções para Compilação

Para executar o código do trabalho e disponibilizado no github, você deve realizar os seguintes passos em qualquer terminal de preferência:

1. Acesse a pasta raiz (Estrutura-Dados-TP0) do projeto, onde está presente o Makefile.
2. Execute o comando make.
3. Acesse a pasta “bin” do projeto.
4. Inicie o executável “run” considerando as seguintes flags.
    - 4.1. **[Obrigatória]** -i “caminho/nome_arquivo.ppm”
        - 4.1.1. Informe o caminho e nome do arquivo de imagem do tipo .PPM que será convertido
    - 4.2. **[Obrigatória]** -o “caminho/nome_saida”
        - 4.2.1. Informe o caminho e nome do arquivo resultante após a conversão da imagem para tons de cinza.
        - 4.2.2. **Observação:** Não é necessário informar a extensão .PGM no nome do arquivo de saída.
    - 4.3. **[Opcional]** -p [“caminho/nome_arquivo_log.out”]
        - 4.3.1. Ao informar está flag você pode definir o caminho e nome do arquivo “.OUT” gerado no fim da execução.
        - 4.3.2. Por padrão será criado um arquivo chamado “log.out” no mesmo local do executável.
    - 4.4. **[Opcional]** -l
        - 4.4.1. Ao informar está flag, você estará habilitando os registros de acesso a memória feitos pelo “memlog.h”.
        - 4.4.2. Ao **não** informar está flag, será gravado apenas o tempo de execução do programa.
5. Após informar as flags obrigatórias e as opcionais que deseja, um arquivo .PGM deve ser criado no local e com o nome informado no parâmetro -o.
6. O arquivo gerado deve representar a mesma imagem informada pelo parâmetro -i, porém deve estar em tons de cinza.