Trabalho Pratico 0
Conversão de imagem PPM para PGM

Leonardo de Oliveira Maia: 2019042139

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

lom2019@ufmg.br

1. Introdução

O problema proposto foi implementar um programa que recebe como argumento uma imagem PPM e gera uma imagem em formato PGM. A partir da transformação dos pixels RGB(red, green, blue) da imagem PPM para a escala cinza do formato PGM, estes pixels devem ser armazenados em alguma estrutura para a criação de um arquivo com as especificações do formato PGM.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

2.1. Estrutura de Dados

A implementação do programa teve como base o uso de loops que percorrem o arquivo da imagem PPM passada como argumento do programa e preenche duas variáveis do tipo ponteiro com nomes “imagem_PPM” e “imagem_PGM” que, de acordo com o tamanho das imagens (informado no arquivo da imagem PPM), seria alocado o espaço necessário nestas. Foi feito o uso apenas de um vetor para armazenar as informações necessárias de cada imagem, pois para alocar em uma matriz seria necessário fazer a alocação dinâmica de várias linhas ou colunas e para ler seria necessário o uso de 2 loops para percorrer a variável, consumindo mais espaço e tempo.

Essa estrutura de dados foi montada numa classe fazendo o uso de 3 funções, sendo que, toda vez que foi feito uma leitura ou o preenchimento de uma variável, sempre foi feito o uso apenas de um loop para cada procedimento, fazendo com que o custo para o programa funcionar fosse baixo.

Além das 3 funções de leitura do arquivo, transformação do arquivo de PPM para PGM e a criação do arquivo no formato PGM na pasta tmp do Linux, o construtor da classe GreyScale que transformara as informações do arquivo que o programa recebe para o formato que queremos, fazendo o programa ser mais eficiente na transformação sem a necessidade de uma nova função.

2.2. Classes

Para modularizar a implementação, foram montadas três classes principais. As duas classes mais simples, porém, importantes para o funcionamento do programa são as classes “RGB” e “GreyScale”, nestas serão armazenadas as informações de cada pixel das imagens PPM e PGM. A classe mais importante, chamada “Imagem”, armazena as duas

variáveis do tipo ponteiro com informações sobre o arquivo de entrada no formato PPM (ponteiro que aponta para um vetor feito da classe RGB) e o arquivo de saída no formato PGM (ponteiro que aponta para um vetor feito da classe GreyScale), nesta classe será feito a leitura do arquivo PPM, armazenamento, e criação do arquivo PGM, isso tudo fazendo uso apenas de loops e funções presentes nas classes simples de pegar o valor de seus atributos.

2.3 Leitura e Armazenamento

Para a leitura, inicialmente verifica se existe algum arquivo em formato PPM para realizar a leitura de suas informações. Após verificado, para cada elemento do arquivo aberto em uma função da classe “Imagem” verifica a consistência da informação para armazenar dentro da variável imagem_PPM com espaço ideal alocado.

Feito a alocação com as informações adicionadas, inicia o processo de conversão de imagem, que realiza uma leitura no vetor de imagem_PPM e replica com a adaptação dos atributos para imagem_PGM.

Por fim, será feito a leitura da variável imagem_PGM e criado um arquivo na pasta tmp do Linux em formato PGM com as informações lidas.

Isto tudo será feito fazendo o uso da biblioteca de leitura de arquivos “fstream”.

3. Analise Complexidade

Observando as funções de leitura e armazenamento, percebemos que as funções apresentam a quantidade de interações igual a

m(altura).n(largura)

estas interações ocorrem nas funções da classe “Imagem” e todas tem a mesma complexidade que seria da ordem $O(n^2)$

Analisando o espaço, como será armazenado m.n pixels em cada estrutura, temos que em cara variável a complexidade será de $O(n^2)$. Portanto, a ordem será de $O(n^2)$.

4. Estratégias de Robustez

Foi feito o uso de asserts com o auxilio da biblioteca msgassert que ao longo do programa foi checado questões relacionadas a:

-Consistência da imagem com seus argumentos

-Abertura correta do arquivo

-Passagem de argumentos corretos para a iniciação do programa

5. Análise Experimental

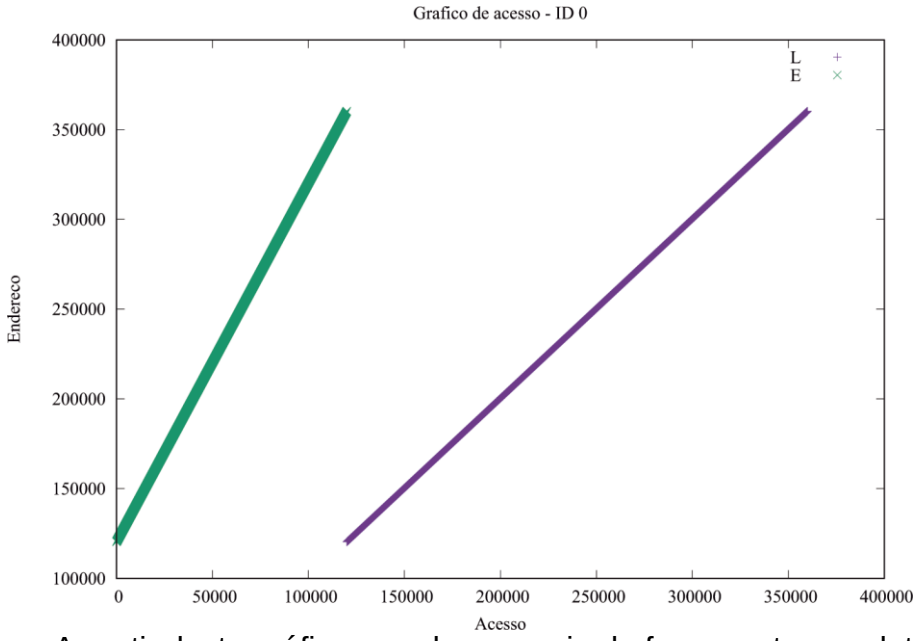
Para a analise experimental foi feito o uso do da biblioteca “memlog” para calcular o tempo e acesso de endereço ao longo das funções.

A analise a seguir foi feito o uso da imagem bolao.ppm disponibilizada como exemplo:

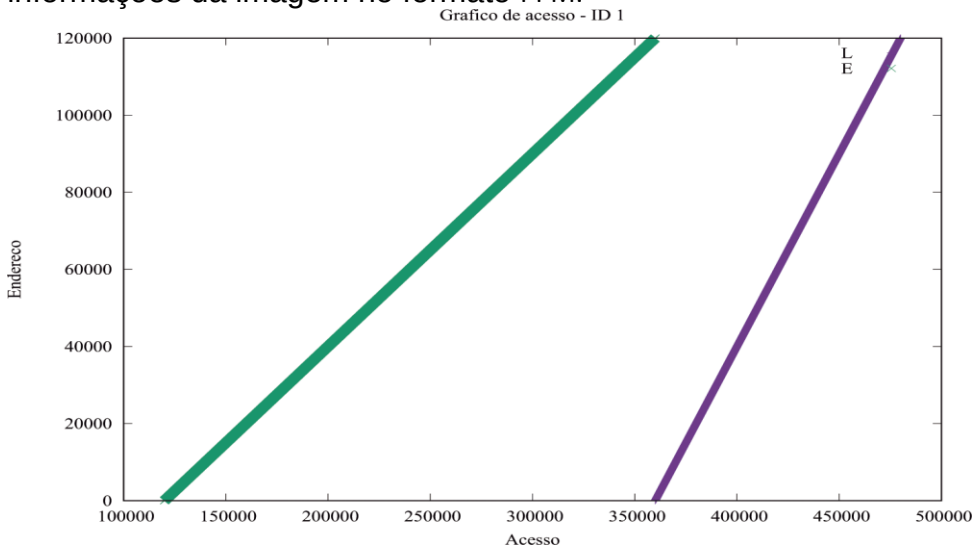
log.out

```
I 1 24553.886695077
E 0 2 0 0.000985069 140439215841296 8
E 0 3 0 0.000987735 140439215841312 8
E 0 4 0 0.000988284 140439215841328 8
E 0 5 0 0.000988627 140439215841344 8
E 0 6 0 0.000988926 140439215841360 8
E 0 7 0 0.000989226 140439215841376 8
E 0 8 0 0.000989536 140439215841392 8
E 0 9 0 0.000989845 140439215841408 8
E 0 10 0 0.000990159 140439215841424 8
E 0 11 0 0.000990550 140439215841440 8
E 0 12 0 0.000990879 140439215841456 8
E 0 13 0 0.000991179 140439215841472 8
E 0 14 0 0.000991474 140439215841488 8
E 0 15 0 0.000991777 140439215841504 8
E 0 16 0 0.000992087 140439215841520 8
```

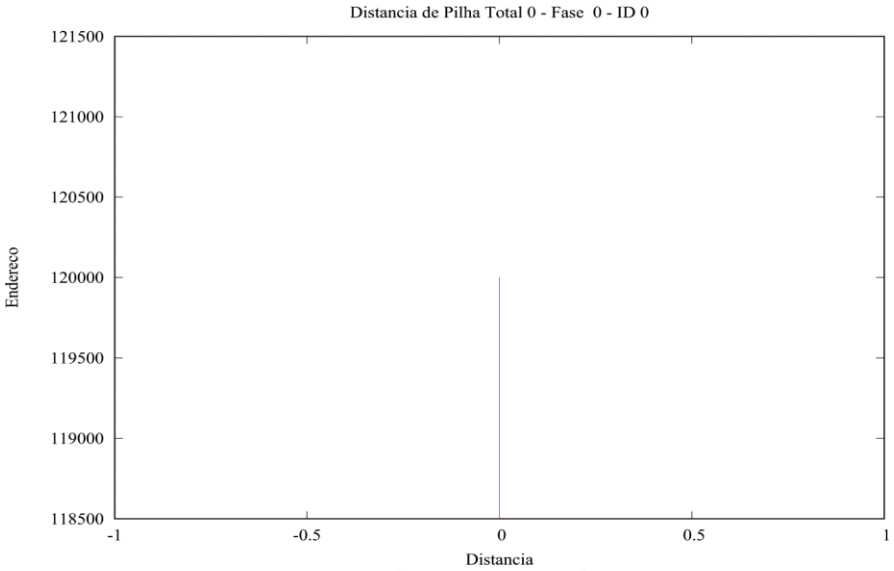
Ao analisarmos esta imagem, percebemos que o tempo decorrido ao longo de um loop está próximo de 0.0000003segundos e isto se mantem ao longo do código, já que toda função tem a mesma ordem de complexidade.



A partir deste gráfico gerado por meio da ferramenta gnuplot, vemos o padrão de acesso a memoria ao longo do tempo onde se lê as informações da imagem no formato PPM.



E este gráfico, representa o acesso a memoria ao longo do tempo com as informações da imagem no formato PGM, vemos que se mantem um padrão linear, sendo algo esperado já que a complexidade se mantém a mesma.



E por fim temos o gráfico para análise da distância dos locais acessados em relação aos próprios acessos. Como observado, a distancia se iguala a 0 e isto se deve ao fato de que cada fase possui um registro de pilha própria, como somente será acessado uma vez por fase, a distância sempre será 0, valendo para todas as fases esta distância e não sendo necessário mostrar outros gráficos que terão a mesma distância.

6. CONCLUSÕES

Podemos concluir que o trabalho está consistente no que se diz respeito a sua complexidade em suas funções e o gráfico de acesso de endereços ao longo do tempo. Está funcional em respeito a criação de imagem em formato PGM e não está permitindo falhas com o uso de asserts.

Teve algumas dificuldades para a criação de funções para a leitura correta dos arquivos, mas após ter feito a leitura correta, não houve dificuldades para transformar a imagem PPM para PGM ou criar o arquivo no formato desejado.

O programa está muito eficiente em questão de desempenho devido ao uso de funções simples somente para leitura e preenchimento.

Ao longo do desenvolvimento percebe-se a importância do uso de boas praticas para identificar problemas prejudiciais ao código rapidamente e fácil adaptação do código para permitir novas estruturas.

7. Bibliografia

-http://qiinformatica.blogspot.com/2017/04/desenvolvendo-um-leitor-de-imagens-ppm.html
-https://youtu.be/6byB1jX3Fnk
-Manual do Usuário: Analisamem

8. Instruções para compilação e execução

Ao abrir o programa, será necessário colocar um arquivo em formato .ppm na raiz do projeto TP.

Basta escrever no terminal “make all” que será compilado o programa e gerado os arquivos .o na pasta “obj” e um .out na pasta “bin”.

Escreva no terminal “./bin/run.out -i (1) -o (2) -p (3) -l”

Sendo que:

1- Nome da imagem .ppm
2- Nome do arquivo com final .pgm criado no formato .pgm
3- Nome do arquivo para analise do desempenho do código

O programa irá gerar a imagem na pasta tmp do Linux com o nome informado em 2 e um arquivo para analise de desempenho na raiz do projeto.

Para limpeza dos arquivos objetos e do .run, basta escrever “make clean” no terminal.