Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)

Lorenzo Carneiro Magalhães

2021031505

lorenzocarneirobr@gmail.com

TRABALHO PRÁTICO 0

Belo Horizonte - MG - Brasil

15/09/2022

SUMÁRIO

1. **INTRODUÇÃO**

O trabalho prático consistiu em implementar um algoritmo para a conversão de arquivos no formato .ppm para o formato pgm e calcular o custo computacional final através de uma análise de complexidade do tempo e do espaço. Os resultados do trabalho constam nesse documento, o qual detalha o funcionamento do algoritmo e a complexidade dele.

2. **MÉTODO**

2.1. **Especificações da máquina utilizada**

A máquina utilizada possui 16 gigabytes de memória RAM e processador i7-1165G7 2.80GHz com frequência máxima de 4.70GHz (64 bits CPU).

O programa foi desenvolvido na linguagem C++, utilizando o editor de texto Visual Studio Code e o sistema operacional Kubuntu, e compilado utilizando o G++.

2.2. **Estrutura do código**

É importante ressaltar que todos os códigos seguem os princípios da engenharia de software.

O código foi dividido em 3 módulos de código: RGB, Matrix e PPMHandler. Cada módulo possui função de, respectivamente: criar uma estrutura para armazenar os valores RGB; criar uma matriz dinâmica com valores do tipo RGB e métodos para sua manipulação; lidar com o arquivo no formato .ppm para a conversão para o formato .pgm.

2.3. **Detalhamento da implementação**

Para a implementação do programa e do algoritmo de conversão do arquivo, o código foi feito otimizando a memória utilizada. Segue os detalhes da implementação nos próximos subtópicos.

2.3.1. **Classes**

Para modularização do código e uma melhor abstração do código, foram criadas 2 classes: RGB e Matrix.

A classe RGB, como dito anteriormente, tem função apenas de armazenar os valores respectivos à intensidade de vermelho, verde e azul do pixel correspondente. Para isso, foi usado o tipo uint8_t para otimizar a utilização de memória, já que suporta apenas 1 byte, ou seja, 8 bits, que é o valor mínimo necessário para o armazenamento dos componentes de cor na forma RGB, visto que os valores da estrutura variam de 0 a 255.

Também foram implementadas outras funções para a manipulação e construção da classe.

Já a classe Matrix é responsável pela criação da estrutura responsável pelo armazenamento da estrutura RGB de cada pixel. Sendo assim, a matriz é alocada dinamicamente com as dimensões da imagem.

A escolha de declarar variáveis para o número de linhas e colunas foi realizada em razão da velocidade de processamento maior em detrimento do espaço relativamente pequeno ocupado por tais variáveis. A classe também contém funções e métodos para a manipulação dos valores e da estrutura, além de um destrutor que desaloca a memória antes alocada.

2.3.2. **Manipulação do arquivo**

O código relativo à manipulação do arquivo é simples e seguro, já que verifica as especificações da imagem e se o arquivo está aberto, visando evitar possíveis erros.

As variáveis durante essa manipulação são do tipo unsigned, size_t ou string, dependendo da necessidade da operação. Possivelmente a manipulação pode ser otimizada, mas não foram achados meios para tal otimização de maneira a manter o código fielmente modularizado e de fácil abstração.

3. **ANÁLISE DE COMPLEXIDADE**

Observando o código, percebe-se que as funções escritas no módulo *PPMHandler.cpp* são as mais complexas, já que envolvem todas as estruturas dos outros módulos e realiza a função principal do programa, que é ler um arquivo *.ppm* e convertê-lo para *.pgm*.

Visivelmente, a complexidade das funções nesse módulo são de ordem quadrática, ou seja, pertencem a **$O(n^2)$**. Isso ocorre devido ao fato de iterar **m * n** vezes sobre a matriz que guarda os pixels, tomando **m** como a altura e **n** como o comprimento da imagem em pixels. Esse padrão de iteração ocorre nas funções: *readPPM* e *turnBW*(que está inclusa na função *writePGM*).

Em relação ao espaço, como descrito anteriormente, o programa armazena todos os pixels da imagem, resultando em m * n espaços de memória alocados, fazendo com que seja de ordem quadrática também.

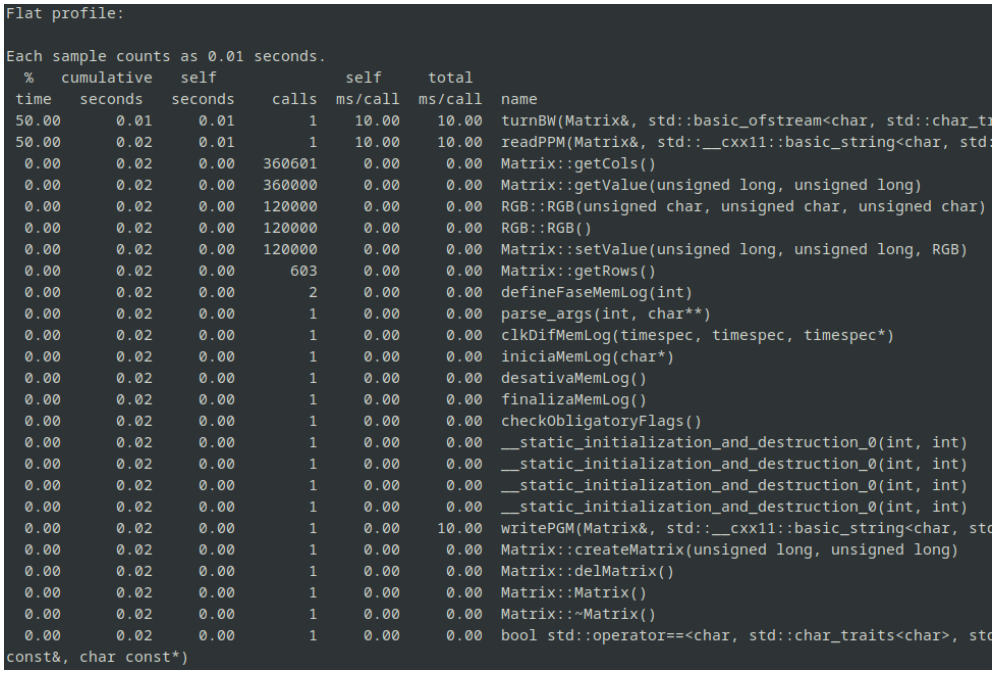
4. **ESTRATÉGIAS DE ROBUSTEZ**

A implementação de asserts, da biblioteca msgassert, foi implementada ao longo do problema, para evitar eventuais erros na execução do programa. Alguns dos pontos verificados são:

- especificações da imagem lida
- abertura adequada do arquivo
- inicialização inadequada da matriz
- checagem de flags obrigatórias

5. **ANÁLISE EXPERIMENTAL**

Para a análise de complexidade temporal, foram utilizadas 2 ferramentas para complementar a análise do código: gprof e a biblioteca analisamem.



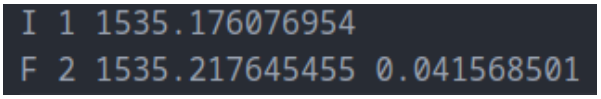
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 50.00      0.01     0.01        1    10.00    10.00  turnBW(Matrix&, std::basic_ofstream<char, std::char_t
 50.00      0.02     0.01        1    10.00    10.00  readPPM(Matrix&, std::__cxx11::basic_string<char, std
  0.00      0.02     0.00   360601     0.00     0.00  Matrix::getCols()
  0.00      0.02     0.00   360000     0.00     0.00  Matrix::getValue(unsigned long, unsigned long)
  0.00      0.02     0.00   120000     0.00     0.00  RGB::RGB(unsigned char, unsigned char, unsigned char)
  0.00      0.02     0.00   120000     0.00     0.00  RGB::RGB()
  0.00      0.02     0.00   120000     0.00     0.00  Matrix::setValue(unsigned long, unsigned long, RGB)
  0.00      0.02     0.00      603     0.00     0.00  Matrix::getRows()
  0.00      0.02     0.00        2     0.00     0.00  defineFaseMemLog(int)
  0.00      0.02     0.00        1     0.00     0.00  parse_args(int, char**)
  0.00      0.02     0.00        1     0.00     0.00  clkDifMemLog(timespec, timespec, timespec*)
  0.00      0.02     0.00        1     0.00     0.00  iniciaMemLog(char*)
  0.00      0.02     0.00        1     0.00     0.00  desativaMemLog()
  0.00      0.02     0.00        1     0.00     0.00  finalizaMemLog()
  0.00      0.02     0.00        1     0.00     0.00  checkObligatoryFlags()
  0.00      0.02     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.02     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.02     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.02     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.02     0.00        1     0.00    10.00  writePGM(Matrix&, std::__cxx11::basic_string<char, st
  0.00      0.02     0.00        1     0.00     0.00  Matrix::createMatrix(unsigned long, unsigned long)
  0.00      0.02     0.00        1     0.00     0.00  Matrix::delMatrix()
  0.00      0.02     0.00        1     0.00     0.00  Matrix::Matrix()
  0.00      0.02     0.00        1     0.00     0.00  Matrix::~Matrix()
  0.00      0.02     0.00        1     0.00     0.00  bool std::operator==<char, std::char_traits<char>, st
const&, char const*)
```

Essa imagem é referente a um log do gprof executado sobre o programa que converte a imagem *bolao.ppm*, que foi disponibilizada no moodle.

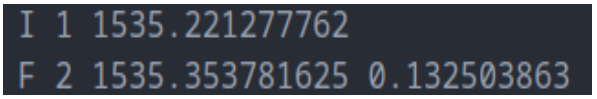
Observando esse perfil de análise, percebe-se que as duas funções que dividem aproximadamente 100% do tempo são a readPPM e a turnBW. Esse fato é esperado, já que são nelas que ocorrem a leitura e a real escrita dos arquivos, em que são executados as outras funções, como a getCols e a getValue diversas vezes.

Por inconstâncias no log do gprof no tempo de cada função, não se pode progredir muito em uma análise temporal se baseando somente nessa ferramenta.



```
I 1 1535.176076954
F 2 1535.217645455 0.041568501
```

perfSnail.out

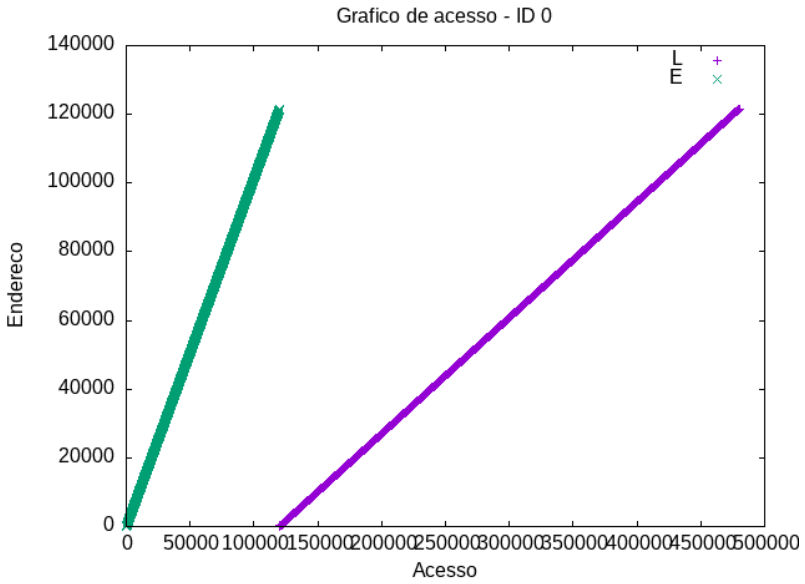


```
I 1 1535.221277762
F 2 1535.353781625 0.132503863
```

perfBlackbuck.out

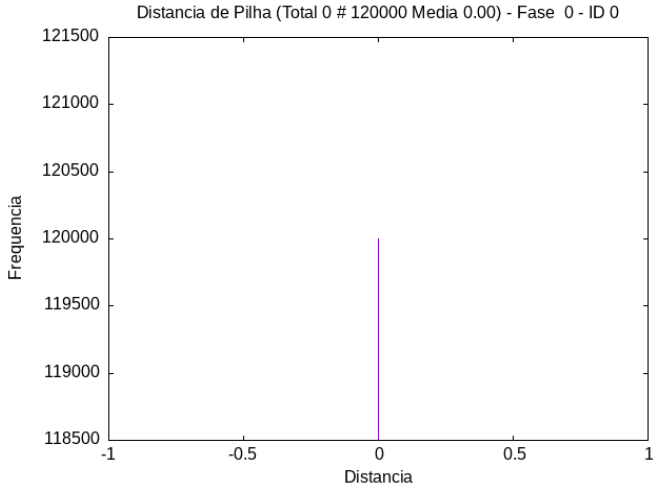
Essas duas imagens, advindas da execução com marcador de tempo do programa, corroboram com a teoria da complexidade pertencer a $O(n^2)$, já que a imagem utilizada para a geração do primeiro log possuía tamanho 256x256 pixels e a do segundo log o dobro, 512x512 pixels. O tempo de execução, que é o terceiro termo da segunda linha, é respectivamente de, 0.04 segundos e 0.13 segundos.

Já em relação ao tempo, foi utilizado a ferramenta gnuplot e biblioteca analisamem.



Esse gráfico representa o padrão de acesso à memória durante o tempo. O gráfico representa bem o que acontece no programa, em que é lida a imagem *.ppm* de entrada e em seguida é escrita uma imagem em preto e branco no formato *.pgm*, que demora mais para ser escrita devido ao fato de acessar outros endereços de memória e realizar operações matemáticas.

Os outros gráficos também são condizentes com o programa, que está dividido em duas fases. Na primeira, há a leitura do arquivo *.ppm* e na segunda há a escrita do arquivo no formato *.pgm* corretamente. Sabendo disso, vamos para uma análise dos gráficos referentes à pilha.



O gráfico apresenta a relação da distância dos locais acessados em relação aos próprios acessos. Percebe-se que há uma distância constante de 0. Isso ocorre devido ao fato de que cada fase possui um registro de pilha própria (conforme a documentação do analisamem). Sabendo que todas as posições da matriz de ID 0 são acessadas apenas uma vez por fase, conclui-se que ela sempre possuirá distância de pilha igual a 0.

A partir dos outros gráficos, é possível fazer análises muito semelhantes e com as mesmas conclusões, portanto não constarão no documento.

6. **CONCLUSÕES**

A análise inicial estava correta, sendo assim provada pelos resultados experimentais. Em relação ao tempo, o programa pertence a $O(n^2)$ e em relação ao espaço também, já que é necessário armazenar m*n pixels. Para os testes realizados, o programa desempenhou satisfatoriamente e apresentou a robustez desejada.

7. **BIBLIOGRAFIA**

ANALISAMEM. Manual do Usuário.
GNU GPROF. Sourceware.
GNU MAKE. GNU.
NETPBM. Wikipedia.

8. **COMPILAÇÃO E EXECUÇÃO**

Para somente a compilação de um arquivo, utilize do comando *make nomeDoArquivo.o*. Todos os arquivos *.o* serão salvos na pasta *obj*.

Para a geração do programa executável, use o comando *make bin*. Desse modo, um arquivo executável será salvo na pasta *bin*. Esse comando também compila os arquivos, se necessário para a criação do executável.

Para executar o arquivo final, basta utilizar o comando *./bin/main*. Para adicionar argumentos à execução, basta digitá-los em sequência ao código acima. Ex.: *./bin/main -i photo.ppm -o photogray.pgm.*

Para a execução da ferramenta gprof, utilize o comando *make gprof*. Esse comando executa o programa com o arquivo escolhido anteriormente e gera um relatório através do gprof.

Para o teste de performance temporal, digite o comando *make perf*. Esse comando irá executar a conversão de um arquivo escolhido anteriormente e gerar um relatório do tempo de execução.

Os relatórios relacionados à memória são gerados através do comando *make mem*.

Para a criação dos gráficos é utilizado o comando *make plot*.

Para a utilização de todas as ferramentas descritas acima, digite o comando *make all*.

Para limpar os arquivos compilados e executáveis, assim como os relatórios de desempenho e saída e as imagens geradas, digite *make clean*.

Todos os relatórios e saídas do programa são armazenados na pasta out, que é criada através do comando *make out*, ou automaticamente a partir da execução dos outros comandos que geram arquivos de saída.

Os outros comandos descritos no *makefile* foram utilizados em testes ou para o desenvolvimento de conteúdo para essa documentação, de modo que possam a não serem mais funcionais, estando ali por fins de documentação do código.

Argumentos disponíveis para a execução:

-i: ao usar, requer a passagem de mais um parâmetro, que corresponde ao nome do arquivo .ppm que será convertido;

-o: ao usar, requer a passagem de mais um parâmetro, que corresponde ao nome do arquivo .pgm que será gerado;

-p: ativa a emissão do relatório de desempenho, sendo necessário a passagem de um parâmetro adicional, que corresponde ao nome do arquivo de log gerado;

-l: ativa o padrão de acesso e localidade.

Uma observação a ser feita é que também foi incluído uma pasta com imagens exemplo. A inclusão delas na pasta do programa possui como intuito fazer com que os comandos não precisem ser alterados no *makefile*, em primeiro momento, para uma execução sem erros.