

Documentação do TP0

Gustavo Guedes de Azevedo Barbosa

1. Introdução

As imagens são elementos presentes na vida e na cultura de praticamente todas as sociedades do mundo. Com o advento da computação e das câmeras fotográficas, passamos a ter uma quantidade de dados referentes a imagens nunca antes imaginada. A manipulação dessas imagens pode ser usada de várias formas, desde filtros para fotos que serão postadas em redes sociais, ao reconhecimento facial utilizado por empresas de segurança.

O objetivo desse trabalho é implementar e aplicar o conceito de TADs, no problema referente às operações de convolução em imagens digitais no formato PGM. Deverão ser implementadas funções para a leitura de uma imagem, operações de convolução e armazenamento de dados pós-processados em um TAD denominado PGM. Esse TAD será testado em um programa que deriva imagens no formato PGM (Portable Gray Map), aplicando um filtro proveniente de uma matriz denominada “kernel de Prewitt”.

Espera-se com isso praticar conceitos básicos de programação em C, além de aprender novas aplicações proporcionadas por tal linguagem.

“Beauty is more important in computing than anywhere else in technology because software is so complicated. Beauty is the ultimate defence against complexity.” -David Gelertner.

2. Implementação

As estruturas de dados utilizadas na implementação foram estruturas de acesso aleatório (indexadas), que podem ser abstraídas como matrizes de índices linha e coluna, respectivamente. Para representar uma imagem PGM foi implementado um struct com os seguintes elementos: número de colunas e número de linhas da imagem, o valor máximo que um pixel pode assumir e uma matriz que representa a imagem.

As funções e os procedimentos implementados foram:

Observação: O termo “matriz”, utilizado ao longo do texto, se refere a ponteiro de ponteiros. Utiliza-se tal terminologia como uma forma de abstração dessa informação.

PGM* CriaPGM (int l, int c, int maximo): Recebe valores inteiros referentes ao número de linhas, número de colunas e valor máximo de um pixel do novo tipo PGM que será criado. Essa função utiliza o comando “malloc” repetidamente para alocar dinamicamente (de acordo com as necessidades do usuário em tempo de execução), um ponteiro do tipo PGM e a matriz que receberá a imagem. Além disso, a função verifica se foi possível fazer tal operação. Caso negativo avisa o usuário e termina o programa. Caso positivo, retorna o ponteiro de PGM devidamente alocado.

PGM* ExpandeImagem (PGM *img): Recebe um ponteiro para o tipo PGM e faz a operação de expansão da imagem contida na estrutura. Essa operação, que é feita para auxiliar o algoritmo de convolução, expande a matriz em duas linhas e duas colunas e replica o valor existente nas colunas exteriores e nas linhas exteriores da imagem recebida como parâmetro, na imagem que será retornada. Para tal operação, utiliza-se novamente o “malloc”.

Ilustração:

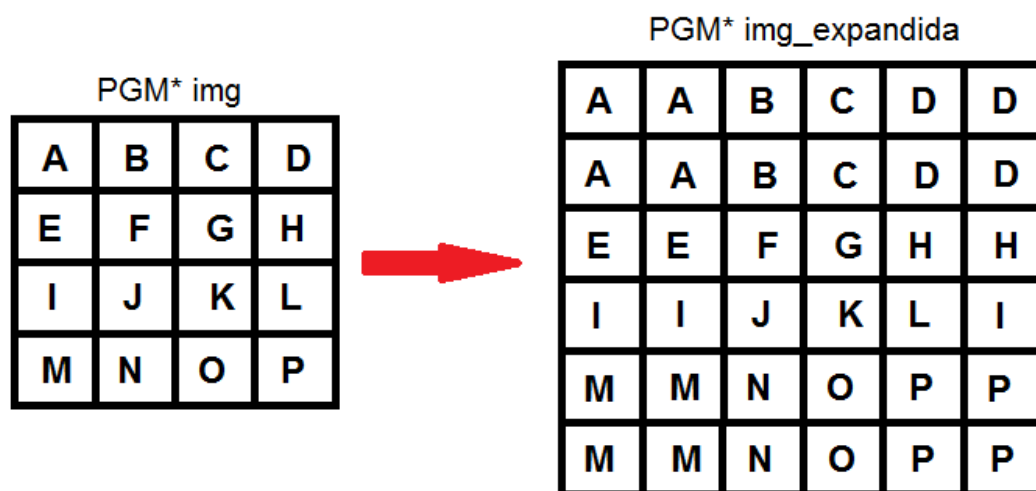


Figura 1

PGM* LerPGM (char *entrada): Recebe uma string (char*) como argumento, sendo o valor dessa variável o nome da imagem que será convoluída. O algoritmo checa se foi possível abrir o arquivo, imprimindo uma mensagem de erro caso negativo. Caso positivo, faz a leitura das primeiras linhas da imagem, que contém o formato de identificação do PGM (“P2”) e logo após, o número de colunas, linhas e o valor máximo do pixel. A partir de então é iniciada a leitura dos pixels da imagem, por meio de uma iteração dupla que salva os valores em variáveis do tipo unsigned char, dentro de um ponteiro do tipo PGM. Tais iterações acontecem “c” vezes e “l” vezes, referentes, respectivamente, ao

número de colunas e linhas da imagem. Há um teste que verifica se esses valores lidos condizem com as dimensões da imagem, imprimindo uma mensagem de erro e fechando o arquivo caso negativo. *É necessária uma observação: a máscara de leitura dos tipos numéricos unsigned char é "%hhu", porém ela não é aceita quando utiliza-se a IDE Code Blocks no Windows e culmina em erros de segmentação e consequentemente, execução.*

char criaKernel ():** Sem receber argumentos, essa função aloca dinamicamente a matriz de operação da convolução, imprimindo mensagem de erro caso não for possível alocá-la. Caso a alocação não apresentar problemas, retorna-se a matriz.

void Convolução (PGM *img, char **kernel, PGM *saida): Recebe um ponteiro de PGM, uma matriz de operações kernel e outro ponteiro de PGM. O primeiro argumento é a imagem lida na função "LerPGM" e expandida na função "ExpandImagem". Isso significa, necessariamente, que o "PGM *img" tem dimensões, referente a "PGM *saida", maiores. A operação de convolução se dá da seguinte forma: são utilizadas 4 iterações, nas quais as duas primeiras são utilizadas para percorrer a matriz que está sendo convoluída, porém, os índices "i" e "j" se comportam de maneira tal que o centro da matriz kernel (elemento k[1,1]) se encontre no interior da "moldura" criada ao realizar a expansão da imagem. Antes do início das duas últimas iterações, uma variável do tipo inteiro é criada para receber o resultado do somatório dos elementos de kernel pela imagem entrada. As duas últimas iterações utilizam-se dos índices "l" e "k", para linha e coluna de kernel. Ambos começam em 1 e decrementam até -1, decisão tomada tendo em vista tanto o fato de que kernel deve ser rotacionada durante a operação, quanto o fato de que, com esses índices, seria mais intuitivo o "movimento" que a operação de atribuição protagonista da função realiza. Onde, para um elemento de "img->imagem[n][m]", tal que n e m são maiores que 0 e menores que "img->l" e "img->c", respectivamente, ao somarmos os índices "l" e "k", poderíamos descrever o seguinte comportamento: escolhe-se arbitrariamente um elemento de "img->imagem", segundo as especificações citadas anteriormente, centra-se nesse elemento o elemento k[1,1] e então multiplica-se os elementos simétricos de kernel pelos elementos da parte (3 por 3) selecionada da imagem e soma-se esses valores.

Logo após a operação descrita, é verificado o valor da variável soma, pois o máximo valor que ela pode assumir é 255 e o mínimo 0 (intervalo de 1 byte de um "char" sem sinal). Finaliza-se a função com a atribuição do resultado das operações do somatório à imagem de saída (PGM *saida), utilizando-se dos índices das duas primeiras iterações, pois o intervalo deles é semelhante à dimensão de "saida->imagem".

PGM* AlocaPGM (PGM*original): Recebe um ponteiro do tipo PGM previamente alocado e retorna um PGM* com alocação de dimensão semelhante, utilizando-se da função já apresentada (PGM* CriaPGM).

void SalvaPGM (PGM* img, char* saida): Recebe um ponteiro do tipo PGM e uma string (char*) que é o segundo argumento ao executar o programa. Esse procedimento é responsável por imprimir em um arquivo, com o nome contido na variável saida, o tipo PGM. Há duas iterações que caminham por “img->imagem” para imprimir o valor dos pixels e uma condição para realizar a quebra de linha quando chega-se no último elemento da coluna.

PGM* DesalocaPGM (PGM* alocado): Recebe um ponteiro do tipo PGM no qual desaloca-se devidamente a matriz “alocado->imagem”, por meio de uma iteração que caminha pelos ponteiros do ponteiro imagem, desalocando-os e logo após, desaloca o próprio ponteiro imagem. Desaloca-se o ponteiro PGM, atribui seu valor a “NULL” para questões de verificação e segurança e então é retornado o ponteiro PGM desalocado.

char DesalocaKernel (char** kernel_alocado):** Segue a mesma linha de raciocínio da parte da função “DesalocaPGM” que desaloca a matriz imagem, porém recebe e retorna tipos diferentes daquela função.

Programa principal (main.c):

A função main recebe argumentos passados ao iniciar o executável, o valor desses argumentos deverá vir na chamada do executável do programa, dentro do seu respectivo diretório, na forma: caso Windows, “foo.exe arg1 arg2”, caso Linux “exec ./foo.exe arg1 arg2”. Nas quais os argumentos devem conter o formato do arquivo, que nesse caso em específico é “*.pgm”. O primeiro passo é declarar as variáveis necessárias para testar a implementação do TAD e dos algoritmos de alocação dinâmica e convolução, que são: a matriz kernel, o PGM* minha_imagem (imagem que o usuário deseja convolver/derivar) e o PGM* saida, que é a variável que irá armazenar o resultado daquela operação. O segundo passo é expandir a imagem lida por meio da função específica. O terceiro passo é realizar as operações sobre as informações coletadas e salvar o resultado em outra imagem. Por fim, desaloca-se todos os espaços de memória utilizados e verifica se essa última etapa ocorreu conforme o planejado, caso positivo imprime uma mensagem de sucesso. Caso negativo imprime uma mensagem de erro e termina o programa.

Organização do código, decisão de implementação e detalhes técnicos:

O código está dividido em três arquivos principais: imagens_PGM.c, imagens_PGM.h, que implementam o Tipo Abstrato de Dados, e main.c implementa o programa principal. Para simplificar a implementação, resolvi criar a

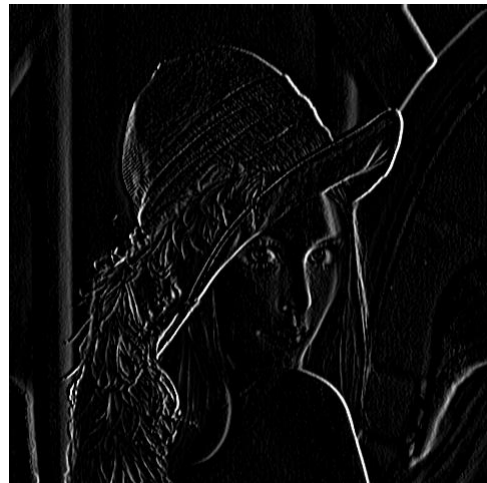
função “PGM* AlocaPGM”, que copia a forma de alocação de um ponteiro PGM para outro de acordo com os argumentos passados para ela, evitando o acesso direto aos elementos da estrutura PGM na main.c. A IDE utilizada para implementar o programa foi Code Blocks 16.01 no Windows 7. Já no Ubuntu 16.04, foi utilizado o compilador GNU gcc v4.5.1. A forma de executar o programa já foi especificada no tópico “Programa principal (main.c)”.

Testes:

O programa foi testado utilizando-se os dois inputs funcionais passados na especificação do Trabalho Prático, “lena.pgm” e “teste.pgm”. Ambos os resultados foram extremamente satisfatórios. Conforme as imagens:



lena.pgm



lenader.pgm

Comando (Windows):

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Gustavo>cd C:\Users\Gustavo\Dropbox\AEDS_II\TP 0\FiltroPGM\bin\Debug
C:\Users\Gustavo\Dropbox\AEDS_II\TP 0\FiltroPGM\bin\Debug>filtropgm.exe lena.pgm
lenader.pgm

**Convolucao feita com sucesso.**

C:\Users\Gustavo\Dropbox\AEDS_II\TP 0\FiltroPGM\bin\Debug>_
```

Onde “filtropgm.exe” é o arquivo executável criado após compilação.

Teste feito em "teste.pgm"

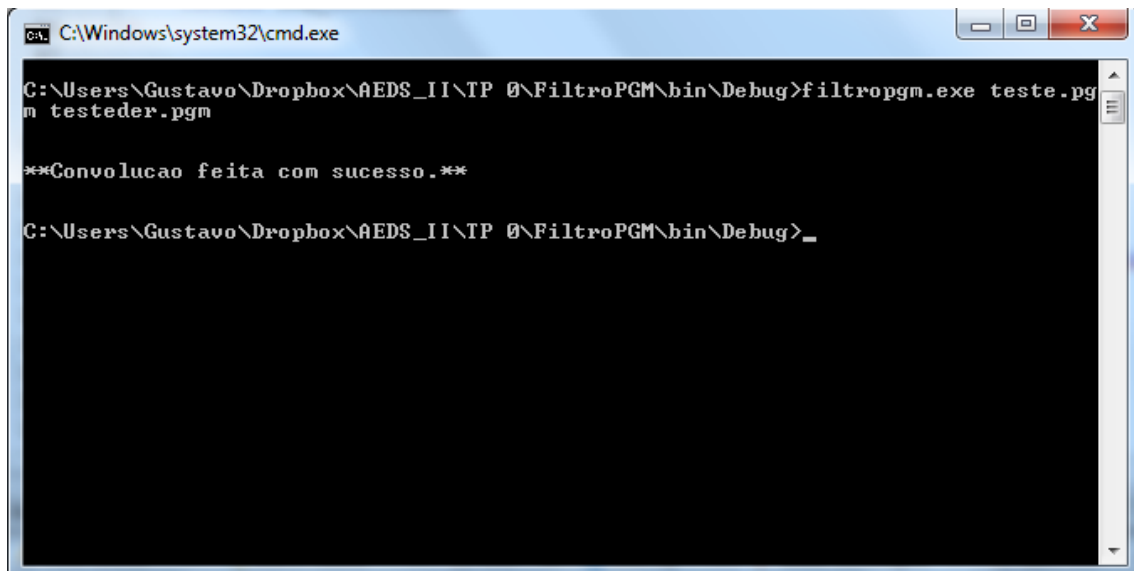
```
1 P2
2 10 10 255
3 92 99 1 8 15 67 74 51 58 40
4 98 80 7 14 16 73 55 57 64 41
5 4 81 88 20 22 54 56 63 70 47
6 85 87 19 21 3 60 62 69 71 28
7 86 93 25 2 9 61 68 75 52 34
8 17 24 76 83 90 42 49 26 33 65
9 23 5 82 89 91 48 30 32 39 66
10 79 6 13 95 97 29 31 38 45 72
11 10 12 94 96 78 35 37 44 46 53
12 11 18 100 77 84 36 43 50 27 59
```

teste.pgm

```
1 P2
2 10 10 255
3 4 255 248 0 0 0 48 23 38 59
4 0 98 218 43 0 0 23 0 43 64
5 0 73 193 73 0 0 0 0 73 89
6 0 43 218 98 0 0 0 0 98 84
7 0 68 98 18 0 0 0 23 43 29
8 4 0 0 0 23 43 18 23 0 0
9 84 0 0 0 148 168 23 0 0 0
10 89 0 0 0 168 168 0 0 0 0
11 64 0 0 0 168 148 0 0 0 0
12 0 0 0 48 143 123 0 23 0 0
```

testeder.pgm

Comando (Windows):



```
C:\Windows\system32\cmd.exe

C:\Users\Gustavo\Dropbox\AEDS_II\TP 0\FiltroPGM\bin\Debug>filtropgm.exe teste.pgm
m testeder.pgm

**Convolucao feita com sucesso.**

C:\Users\Gustavo\Dropbox\AEDS_II\TP 0\FiltroPGM\bin\Debug>_
```

Conclusão:

A implementação do trabalho transcorreu sem maiores problemas e os resultados obtidos foram extremamente satisfatórios. A única grande dificuldade encontrada foi no fato de não existir a máscara "%hhu" no compilador usado pela IDE Code Blocks v16.01 e a máscara "%u" dar problemas de segmentação. Para solucionar o problema, utilizei o Ubuntu 16.04 para compilar o programa utilizando a máscara correta para o tipo "unsigned char".

Anexos:

Listagem dos programas

-imagens_PGM.h

-imagens_PGM.c

-main.c