MAC0110 – Introdução à Computação

BCC IME/USP 2022

Terceito Exercício–Programa – Data de entrega: até 13/07/2022

1 Introdução

Neste EP, iremos implementar e testar alguns algoritmos simples de processamento de imagens.

Processar imagens consiste em aplicar certas transformações que visam facilitar a extração de informações relevantes para um determinado fim.

A figura 1 mostra, à esquerda, uma imagem de (supostamente) células. Ao lado dela são mostradas mais três imagens que são transformações aplicadas sucessivamente à primeira. São elas a limiarização, inversão e rotulação. O objetivo final desse processamento pode ser, por exemplo, calcular o tamanho médio das células.

Figura 1: Da esquerda para a direita: imagem de entrada, limiarização, inversão e rotulação.

Imagens, do ponto de vista computacional, podem ser vistas como matrizes de dimensão $w \times h$ (largura $\times$ altura). O valor em um ponto da matriz codifica alguma informação referente àquele ponto na imagem. Neste EP trataremos apenas imagens monocromáticas (tons de cinza) e suporemos que os valores em cada ponto de uma imagem variam entre 0 e 255 (256 tons de cinza). Os tons de cinza serão chamados de *intensidade*.

Porém, à medida que as imagens são transformadas, esses valores podem ser alterados, e dependendo da transformação os seus valores podem significar outra coisa. Por exemplo, na imagem mais à direita da figura 1, os valores associados a cada ponto identificam um componente conexo e não tem relação com a "cor" do objeto. Na rotulação de componentes conexos, um valor distinto é atribuído a cada componente conexo, e os valores dos componentes variam de 1 a n (n neste caso é o número total de componentes na imagem). O valor 0 é reservado para respresentar o fundo (*background*) da imagem. Note que nesse tipo de imagem, o valor de um ponto pode ser maior que 255. A figura 2 mostra a imagem rotulada com uma coloração artificial, para evidenciar a identificação dos componentes conexos.

Figura 2: Da esquerda para a direita: imagem binária, componentes conexos rotulados, e componentes conexos com cores aleatórias.

2 Visão geral do EP

Neste EP, iremos trabalhar com leitura/escrita de arquivos, alocação dinâmica de memória, manipulação de matrizes, além de outros elementos de programação já trabalhados nos EPs anteriores.

A estrutura geral do programa a ser implementado é a seguinte:

```
Enquanto não é para parar
   Exibir menu de opções e ler a opção escolhida pelo usuário
   Executar a opção do usuário
```

O usuário poderá escolher uma dentre as seguintes opções:

Q `Quit` (terminar o programa)

L `Ler` um arquivo de imagens

S `Salvar` a imagem em arquivo

M `Manter` a imagem-entrada anterior

B `Binarizar` a imagem-entrada

C Calcular `Contorno` da imagem-entrada

F `Filtrar` a imagem-entrada

I `Inverter` a imagem-entrada

R `Rotular` a imagem-entrada

O EP foi especificado pensando-se que a qualquer momento de sua execução são possíveis uma das três seguintes situações:

1. Nenhuma imagem alocada na memória do computador
2. Apenas uma imagem (imagem-entrada I) alocada na memória do computador
3. Duas imagens, uma imagem-entrada I e uma transformação R dela, alocadas na memória do computador

A situação 1 ocorre no início da execução. Nessas condições, apenas as opções `Sair` e `Ler` podem ser executadas.

A situação 2 ocorre após as opções `Ler` ou `Manter`. No caso da opção `Ler`, quaisquer imagens que estejam alocadas devem ser descartadas antes da leitura. No caso da opção `Manter`, a imagem-entrada deve ser mantida e a imagem resultado, caso exista, deve ser descartada.

A situação 3 ocorre após as opções `Binarizar`, `Contorno`, `Filtrar`, `Inverter` ou `Rotular`. Essas cinco transformações são sempre aplicadas sobre a imagem-entrada I e produzem o resultado aqui referenciado como R. Note que, a não ser que o usuário tenha escolhido anteriormente a opção `Ler` ou a opção `Manter`, a imagem-entrada deve ser o resultado da última transformação executada. Além disso, essas cinco operações não fazem sentido na situação 1 acima.

A opção `Salvar` deverá gravar a imagem R caso ela esteja alocada, e em caso contrário deverá gravar a imagem-entrada. Note que a opção `Salvar` não faz sentido na situação 1 acima.

Em situações nas quais as imagens são descartadas, o espaço de memória alocado para as imagens deve sempre ser liberado. Mais detalhes adiante.

3 Detalhes do EP

3.1 Arquivos de imagens

O formato de arquivo que iremos utilizar neste EP é o `.pgm` tipo P2, que é um formato texto (fácil de ler)[1]. Um exemplo é apresentado a seguir:

```
P2
10 10
255
0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 1 1 1 0
1 1 0 0 1 0 0 1 1 0
0 0 1 0 0 1 0 0 1 0
0 0 1 0 0 1 0 0 1 0
0 0 0 1 0 1 0 0 1 0
0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 1 1 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0
```

As três primeiras linhas são de cabeçalho. Eventualmente o cabeçalho pode conter mais linhas (começando com o caractere `#`), que são ignoradas. Neste EP, iremos simplificar e supor que o cabeçalho consiste apenas das três linhas. A segunda linha indica que trata-se de uma imagem 10×10 (largura $\times$ altura). O valor 255 indica o valor máximo que poderíamos ter na imagem. Na prática ele é usado para fazer ajustes na hora da visualização. Iremos usar 255 mesmo que o máximo não seja esse (como no exemplo acima). Após o cabeçalho, seguem os valores dos 10×10 pontos da imagem.

Para a leitura das primeira e terceira linhas podemos usar

```
fscanf(f, "%[^\n]\n", line);
```

onde `f` é uma variável do tipo `FILE *` e `line` é um vetor do tipo `char` com tamanho suficientemente grande. Esse comando simplesmente lê, a partir da posição atual, uma sequência de caracteres terminada por `\n` O efeito prático, no nosso caso, é ler uma linha sem nos preocuparmos com o conteúdo dela.

Para escrever uma imagem em um arquivo, basta escrever o cabeçalho conforme acima e em seguida os valores da matriz que corresponde à imagem que queremos gravar. Por exemplo, para gravar um número inteiro `a` no arquivo cujo descritor está em `f`, basta fazermos

```
fprintf(f, "%d", a);
```

Protótipo das funções para leitura e gravação de imagem:

[1] Existem vários formatos de arquivo para armazenamento de imagens. A maioria deles usa um formato binário e não texto.

```
/* Leitura de arquivo PGM P2
   Parâmetros:
       (IN)  char * : nome de arquivo (string)
       (OUT) int * : largura da imagem
       (OUT) int * : altura da imagem
   Devolve:
       int * : imagem lida do arquivo (vetor)
*/
int *load_image_from_file(char *filename, int *w, int *h) ;

/* Salvar imagem em arquivo
   Parâmetros:
       (IN) char * : nome de arquivo (string)
       (IN) int  * : imagem (vetor)
       (IN) int : largura da imagem
       (IN) int : altura da imagem
   Devolve:
       Nada (vamos supor que a escrita irá sempre funcionar)
*/
void save_image_to_file(char *filename, int *I, int w, int h) ;
```

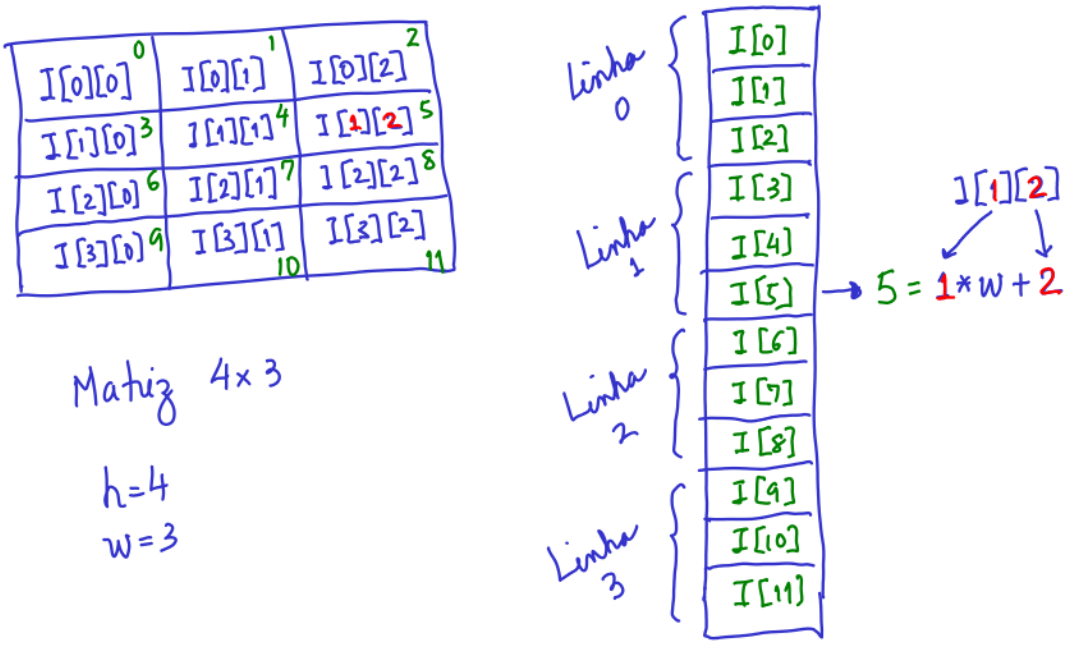
3.2 Alocação dinâmica de memória

Embora estejamos usando a declaração

```
int  M[MAX][MAX] ;
```

nos exercícios vistos em sala de aula, na prática não convém usarmos matrizes desta forma. Vocês poderão verificar que programas que declaram, dessa forma, matrizes muito grandes ou várias matrizes simplesmente não funcionam (pois o espaço ocupado pela matriz ultrapassa o tamanho da pilha de execução).

Para não termos esse problema, mas também para utilizar apenas o tanto que é necessário, neste EP utilizaremos alocação dinâmica de memória. Uma matriz pode ser pensada como um bloco de memória. A forma mais simples é pensar em um bloco linear de memória (vetor) e mapear as coordenadas de uma matriz no índice correspondente nesse vetor. Veja no exemplo a seguir como uma matriz com 4 linhas e 3 colunas pode ser mapeada em um vetor de 12 posições. Em destaque está o elemento $I[1][2]$ que no vetor corresponde ao elemento que ocupa a posição de índice 5.



Ou seja, iremos armazenar imagens em vetores e fazer a interpertação conforme o ilustrado acima. Para alocar vetores em C, no nosso caso para armazenar uma imagem de dimensão `w x h`, fazemos como segue:

```
#include <stdlib.h>

int *I ;

I = malloc(sizeof(int)*w*h) ;
```

O comando acima aloca um bloco com "tamanho de um `int`" vezes `w` vezes h bytes e atribui o endereço do início desse bloco em `I`. (Note que poderíamos usar uma matriz de `unsigned char` (byte) ao invés de um matriz inteira, mas como o número de componentes conexos em uma imagem pode ser maior que 255, usaremos o tipo `int` por conveniência.)

Ao final, deve-se liberar a memória alocada, da seguinte forma:

```
free(I) ;
I = NULL ;
```

Observe que só faz sentido aplicar o comando `free(I)` quando `I` é uma variável do tipo apontador que contém o endereço de uma porção de memória que foi e está alocada com o uso da função `malloc()`. Por praxe, é recomendável atribuir `NULL` sempre que liberamos a memória alocada, para não corrermos o risco de termos apontadores "soltos" no meio do programa.

3.3 Transformações a serem implementadas

Seja I uma imagem de dimensão $w \times h$. Todas as funções aqui especificadas deverão alocar e devolver, via `return`, uma imagem R de mesma dimensão da imagem de entrada I. A imagem de entrada I deve continuar intacta, sem alterações.

Inversão: Simplesmente calculamos o complemento de $I[i][j]$, $i = 0, \ldots, h-1$ e $j = 0, \ldots, w-1$, com respeito ao máximo 255; isto é, o valor invertido de um ponto (i, j) é dado por $255 - I[i][j]$.

```
/* Inverter uma imagem
   Parâmetros:
       (IN) int * : imagem (vetor)
       (IN) int : largura da imagem
       (IN) int : altura da imagem
   Devolve:
       int * : imagem invertida (vetor)
*/
int *invert_image(int *I, int w, int h) ;
```

Binarização (por limiarização): Dados t, $t \in [0, 255]$, a limiarização de I por t é definida por, para todo $i = 0, \ldots, h-1$ e $j = 0, \ldots, w-1$,

$$R[i][j] = \begin{cases} 255, & \text{se } I[i][j] \geq t, \\ 0, & \text{caso contrário.} \end{cases} \tag{1}$$

```
/* Binarizar uma imagem
   Parâmetros:
       (IN) int * : imagem (vetor)
       (IN) int : largura da imagem
       (IN) int : altura da imagem
       (IN) int : limiar
   Devolve:
       int * : imagem limiarizada (vetor)
*/
int *binarize(int *I, int w, int h, int t) ;
```

Contorno: O contorno (interno) da imagem I é calculado da seguinte forma: $I[i][j] - F[i][j]$, $i = 0, \ldots, h-1$, $j = 0, \ldots, w-1$, na qual F é o resultado do filtro de mínimo aplicado à imagem I, usando o tamanho de filtro $d = 3$ (veja como funciona esse filtro no próximo item). O exemplo a seguir mostra uma imagem de entrada e o respectivo contorno (ele pode ser obtido seguindo-se a sequência de opções `Ler`, `Contorno`, `Salvar`).

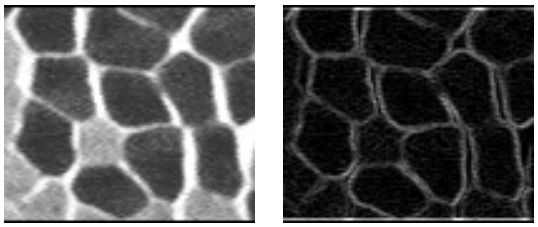


Figura 3: Exemplo de uma imagem e seu respetivo contorno.

Filtragem: Neste EP implementaremos os filtros de mínimo, mediana e máximo para uma vizinhança $d \times d$ (d um inteiro positivo ímpar), que consiste do seguinte. Para cada coordenada (i, j) da imagem, os valores dos pontos na região $d \times d$, centrada no próprio ponto, devem ser ordenados e então aquele valor que ocupa a posição correspondente ao filtro desajado deve ser escolhido como o resultado do filtro no ponto (i, j):

- mínimo: menor elemento na região $d \times d$
- mediana: o elemento central na lista ordenada de elementos na região $d \times d$
- máximo: maior elemento na região $d \times d$

Note que para os pontos na borda da imagem não há todos os vizinhos. Para contornar essa situação, você deve

1. Inicialmente criar uma imagem que é uma cópia da imagem de entrada, porém com a diferença de possuir linhas e colunas adicionais, de forma a garantir que o filtro possa ser aplicado para todos os pontos, mesmo aqueles próximos à borda da imagem. Nessa imagem-cópia alargada haverá linha(s) adicional(is) acima, cujo conteúdo será igual ao da primeira linha da imagem de entrada; similarmente, haverá linha(s) adicional(is) abaixo, cujo conteúdo será igual ao da última linha da imagem original. Analogamente, haverá coluna(s) à esquerda e à direita. Nos cantos da imagem nova, devem ser copiados os valores dos respectivos cantos na imagem original. Por exemplo, se $d = 5$ então será necessário alargar a imagem nos quatro lados por uma largura de dois.
2. Gerar a imagem filtrada com o mesmo tamanho da imagem original (e para isso você pode processar os pontos de interesse sobre a imagem-cópia alargada, sem se preocupar com a existência ou não dos vizinhos).

```
/*
   Aplicar um filtro sobre uma imagem
   Parâmetros:
       (IN) int * : imagem (vetor)
       (IN) int : largura da imagem
       (IN) int : altura da imagem
       (IN) int : tamanho do filtro (inteiro positivo ímpar)
       (IN) int : tipos 1, 2, 3, sendo 1=min, 2=median, 3=max
   Devolve:
       int * : imagem filtrada (vetor)
*/
int *filter_image(int *I, int w, int h, int d, int tipo) ;
```

Rotulação de componentes conexos: Um ponto (i, j) na imagem tem 8 vizinhos. Dois pontos vizinhos com um mesmo valor não nulo fazem parte de um mesmo componente conexo. Um conjunto de pontos na imagem é um componente conexo se todos eles possuem um mesmo valor e se existe um caminho ligando pontos vizinho-a-vizinho entre quaisquer par de pontos no conjunto. Um componente conexo é maximal se não existe outro componente conexo que o contém. A rotulação de componentes conexos identifica todos os componentes maximais em uma imagem, e atribui um mesmo rótulo numérico de identificação a todos os pontos de um componente.

A rotulação é em geral aplicada sobre imagens binárias (com valor 0 e 1 ou 0 e 255). O valor zero na imagem é considerado o fundo e não é de interesse na rotulação. Componentes podem ser pensados como ilhas de pontos conexos com valor 1 (ou diferente de zero). Como dito antes, os componentes são enumerados de 1 a n (sendo n o número total de componentes). Nas situações em que há poucos componentes na imagem, será muito difícil identificá-los por meio de visualização (pois as intensidades próximas de zero correspondem a cores praticamente pretas). Portanto, após calculados os componentes, caso $n <= 127$, os rótulos devem ser multiplicados pela parte inteira de $255/n$.

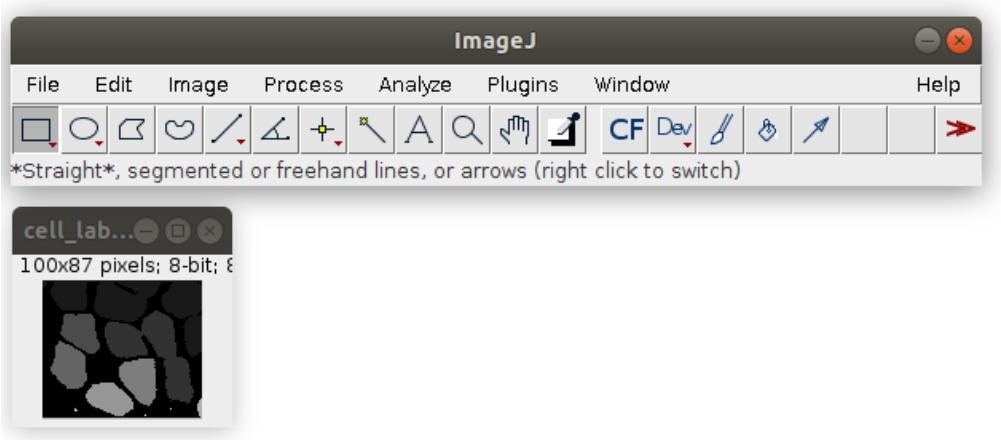
```
/* Rotular componentes conexas
   Parâmetros:
        (IN) int * : imagem (vetor)
        (IN) int : largura da imagem
        (IN) int : altura da imagem
        (OUT) int * : número de componentes conexas
   Devolve:
        int * : imagem com componentes conexas rotuladas (vetor)
*/
int *label_components(int *I, int w, int h, int *ncc) ;
```

4 Como visualizar imagens

Idealmente seria interessante visualizar as imagens à medida que elas são trasnformadas. Porém, C não é uma linguagem que tem fecilidades para isso. Assim, para visualizar as imagens desse EP iremos recorrer a algum visualizador de imagens e que seja capaz de ler o formato que estamos utilizando.

Uma opção online está em https://www.kylepaulsen.com/stuff/NetpbmViewer/.

Outra opção é o software ImageJ (https://imagej.nih.gov/ij/). É um programa em Java, que inclusive pode ser usado para visualizar imagens rotuladas, com cores indicando os componentes, como mostrado no início do enunciado.



Para ver os componentes em cores, após abrir o arquivo, pode-se executar a opção

```
Image ---> Lookup Tables ---> Glasbey
```

OBS.:

1. Pontos não bem esclarecidos e/ou inconsistências serão revistos à medida que se mostrarem necessários.
2. Alguns arquivos de imagens estão disponíveis no e-disciplinas, junto à tarefa EP3.