[PSI3471 aula 2 parte 2. Início]

Crescimento de semente

As imagens usadas nesta apostila podem ser baixadas em Linux executando:

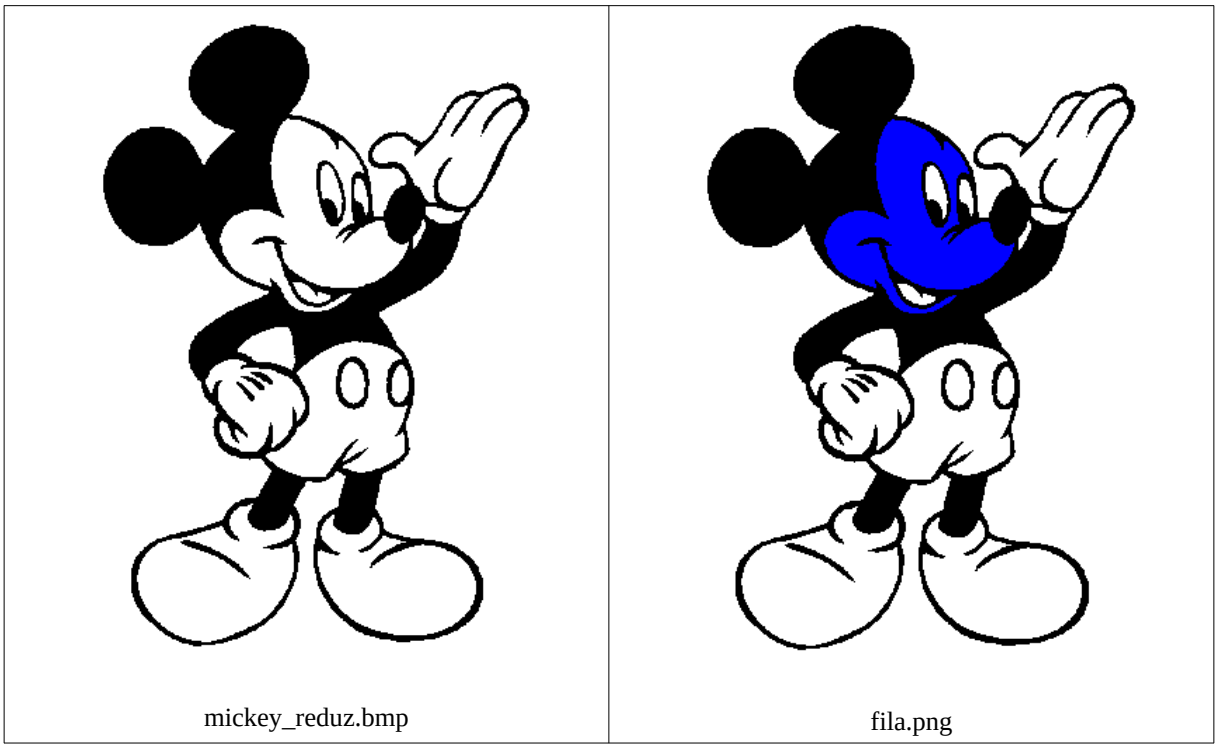
```
>wget -nc -U 'Firefox/50.0' http://www.lps.usp.br/hae/apostila/compcon.zip
>unzip compcon
```

1. O problema

Nesta aula, estudaremos os algoritmos de crescimento de semente ou crescimento de regiões usando fila, pilha ou recursão.

Nota: Não há nada de muito especial com este tópico para processamento de imagens. É simplesmente um assunto que selecionei, entre muitos possíveis, para vocês se familiarizarem um pouco mais com o processamento de imagens.

Considere o problema de pintar a cara do Mickey de azul, dadas as coordenadas (l_s, c_s) de um pixel-semente dentro da cara (figura 1).



mickey_reduz.bmp | fila.png

Figura 1: Dadas as coordenadas de um pixel-semente *s* dentro do rosto do Mickey, queremos pintar de azul todos os pixels brancos 4-conectados a *s*.

Aqui, vamos usar 4-conectividade. Nessa conectividade, dois pixels são considerados vizinhos entre si se compartilharem uma aresta comum (veja a figura 2 abaixo).

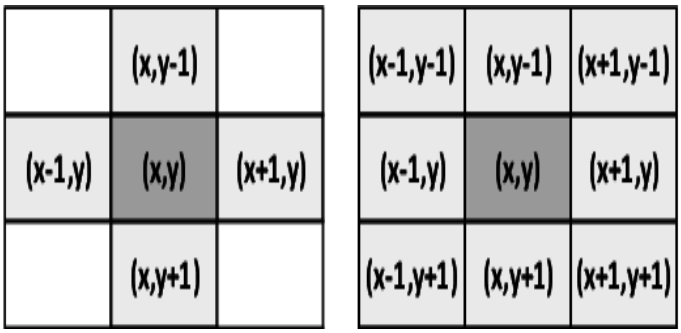


Figura 2: (esquerda) os 4-vizinhos do pixel central, (direita) os 8-vizinhos do pixel central.

Um componente 4-conexo é um conjunto de pixels *foreground* unidos pela 4-conectividade (figura 3). Isto é, dois pixels *foreground P* e *Q* pertencem a um mesmo componente 4-conexo se for ser possível ir de *P* a *Q* passando somente através de pixels 4-vizinhos *foreground*. Na figura 3, há 2 componentes 8-conexos e 6 componentes 4-conexos.

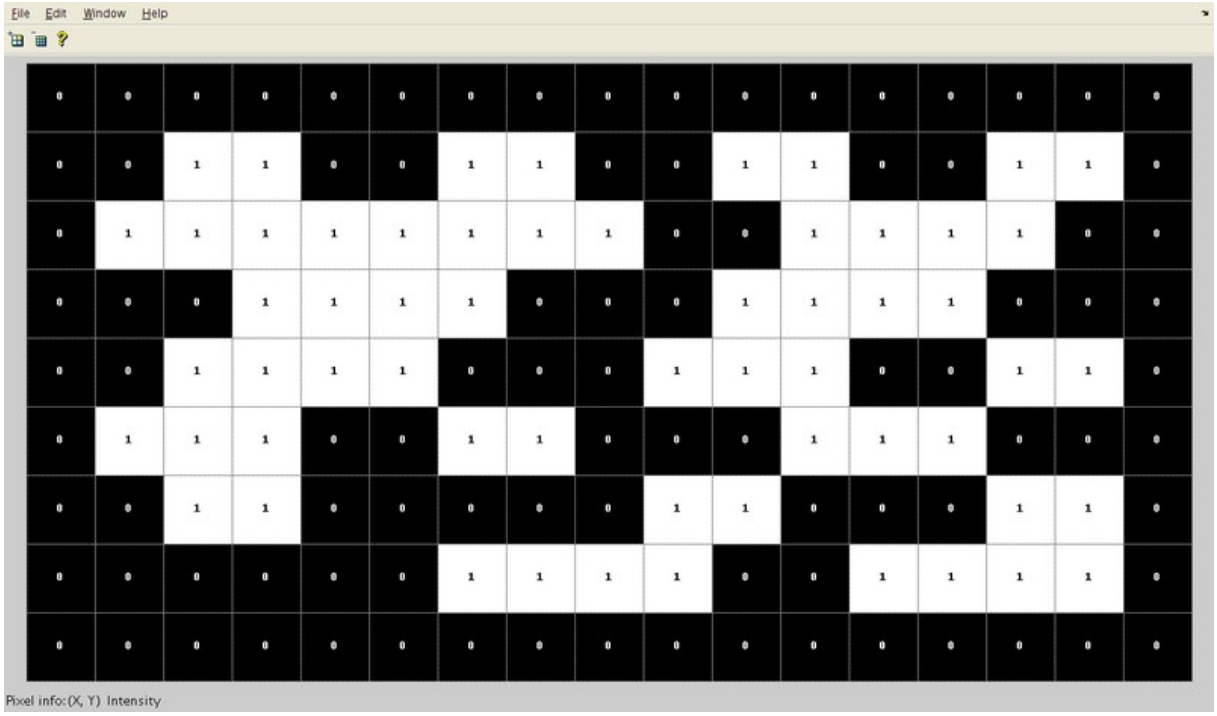


Figura 3: Nesta figura, *foreground*=branco e *background*=preto. Há 2 componentes 8-conexos e 6 componentes 4-conexos.

2. O algoritmo errado

Como resolver este problema? Uma primeira ideia seria o seguinte algoritmo, que denominaremos de “direita, cima, esquerda, baixo”. Vamos começar a andar a partir do pixel-semente (l_s, c_s) numa certa direção, pintando os pixels brancos de azul. Se não der mais para andar naquela direção, por ter encontrado um pixel preto ou azul, vamos mudar de direção.

1) Pinte o pixel-semente de azul.
2) Ande para direita pintando os pixels brancos de azul, até encontrar um pixel preto ou azul.
3) Agora, ande para cima, pintando os pixels brancos de azul, até encontrar um pixel preto ou azul.
4) Ande para esquerda, pintando os pixels brancos de azul, até encontrar um pixel preto ou azul.
5) Ande para baixo, pintando os pixels brancos de azul, até encontrar um pixel preto ou azul.
6) Se passou pelas linhas 2-5 e não conseguiu pintar nenhum pixel de azul, termine o programa. Caso contrário, vá para linha (2) e repita o processo.

Algoritmo 1: “Direita, cima, esquerda, baixo”.

Este algoritmo funciona para o caso da figura 4a, mas falha no caso da figura 4b. Nessa figura, os pixels “X” são pretos, os pixels “.” são brancos, o pixel “1” é a semente e os números representam a ordem em que os pixels brancos foram pintados de azul.

(a)	X X X X X . . X X 1 . X X X X X	X X X X X . . X X 1 2 X X X X X	X X X X X . 3 X X 1 2 X X X X X	X X X X X 4 3 X X 1 2 X X X X X
(b)	X X X X X . . X X 1 . X X X . X X . . X X X X X	X X X X X . . X X 1 2 X X X . X X . . X X X X X	X X X X X . 3 X X 1 2 X X X . X X . . X X X X X	X X X X X 4 3 X X 1 2 X X X . X X . . X X X X X

Figura 4: O algoritmo “direita, cima, esquerda, baixo” funciona no caso (a) mas falha no caso (b).

Existem basicamente dois tipos de algoritmos para resolver este problema: algoritmos de rotulação por escaneamento [Rosenfeld1966, Santiago2019, WikiLabeling] e algoritmos de crescimento de semente ou região (que usam fila, pilha ou recursão) [WikiGrowing]. Nesta aula, veremos somente os algoritmos de segundo tipo, pois são mais versáteis no sentido de que podem ser adaptados facilmente para realizarem outras tarefas.

3. Crescimento de semente

Por que o algoritmo falha na figura 4b? Porque, no pixel “2” tinha dois caminhos possíveis para seguir: para cima e para baixo. O algoritmo resolveu andar para cima. Quando chegou no pixel “4”, esqueceu que poderia ter andado para baixo a partir do pixel “2”. Repare que, num caso mais complexo, podem ter ficado para trás vários caminhos alternativos sem serem explorados.

Como podemos fazer com que o algoritmo “se lembre” de todos caminhos alternativos que não foram explorados? Basta guardá-los todos na memória. Com isso, o algoritmo ficaria assim:

1) Coloque o pixel-semente no conjunto *V* dos pixels a serem visitados.
2) Se o conjunto *V* for vazio, termine o programa.
3) Retire um pixel *p* de *V* e coloque-o no conjunto dos pixels já visitados (com numeração).
4) Olhe a 4-vizinhança de *p* e coloque em *V* os pixels vizinhos que ainda podem ser visitados.
5) Vá para linha 2.

Algoritmo 2: Crescimento de semente usando operações de conjunto.

Uma simulação deste algoritmo está na figura 5. Na linha inferior está o conjunto *V* de pixels a serem visitados (os círculos em vermelho da matriz). Os pixels já visitados recebem a numeração em que foram visitados.

X X X X X X . X X ● . X X X . X X X X X	X X X X X X . X X 1 ● X X X . X X X X X	X X X X X X ● X X 1 2 X X X ● X X X X X	X X X X X X 3 X X 1 2 X X X ● X X X X X	X X X X X X 3 X X 1 2 X X X 4 X X X X X
V={(2,1)}	*V*={(2,2)}	*V*={(1,2) (3,2)}	*V*={(3,2)}	*V*={}

Figura 5: Crescimento de semente guarda os pixels a serem visitados (em círculo vermelho) em alguma estrutuda de dados, por exemplo, num conjunto *V*.

4. Fila

Em vez de usar um conjunto *V* onde a ordem em que os pixels serão retirados do conjunto não está definida, o conjunto de pixels a serem visitados é normalmente implementado usando fila ou pilha. A ordem em que os pixels serão visitados depende de se utiliza fila ou pilha. Para quem não sabe, fila e pilha são duas estruturas de dados muito utilizadas em Computação.

Nota: A ordem em que os pixels serão visitados também depende da ordem de varredura dos 4-vizinhos (por exemplo, NSLO, NOSL, OSLN, etc).

A fila é uma estrutura do tipo first-in-first-out (FIFO). É como uma fila de banco: o cliente que chegou primeiro é atendido primeiro. A figura 6 ilustra o funcionamento de uma fila.

Início do programa	fila vazia
Inserir A na fila.	(fim da fila) A (início da fila)
Inserir B na fila.	(fim da fila) BA (início da fila)
Retirar um elemento da fila: Sai A.	(fim da fila) B (início da fila)
Inserir C na fila.	(fim da fila) CB (início da fila)
Retirar um elemento da fila: Sai B.	(fim da fila) C (início da fila)
Retirar um elemento da fila: Sai C.	fila vazia

Figura 6: Operações numa fila.

Traduzindo o algoritmo 2 para C++ usando fila, temos:

```
 1 //pinta_nocek.cpp 2024
 2 #include <opencv2/opencv.hpp>
 3 #include <queue>
 4 using namespace std; using namespace cv;
 5
 6 Mat_<Vec3b> pintaAzul(Mat_<Vec3b> a, int ls, int cs) {
 7   Mat_<Vec3b> b=a.clone();
 8   queue<int> q;
 9   q.push(ls); q.push(cs); //(1)
10   while (!q.empty()) { //(2)
11     int l=q.front(); q.pop(); //(3)
12     int c=q.front(); q.pop(); //(3)
13     if (b(l,c)==Vec3b(255,255,255)) { //(4)
14       b(l,c)=Vec3b(255,0,0); //(5)
15       q.push(l-1); q.push(c); //6-acima
16       q.push(l+1); q.push(c); //6-abaixo
17       q.push(l); q.push(c+1); //6-direita
18       q.push(l); q.push(c-1); //6-esq
19     }
20   }
21   return b;
22 }
23
24 int main() {
25   Mat_<Vec3b> a=imread("mickey_reduz.bmp",1);
26   Mat_<Vec3b> b=pintaAzul(a,159,165);
27   imwrite("fila.png",b);
28 }
```

```
Algoritmo 2



1) Coloque o pixel-semente no
conjunto V dos pixels a serem
visitados.

2) Se o conjunto V for vazio,
termine o programa.

3) Retire um pixel p de V.

4) Se p for preto ou azul, vá
para (2).
5) Aqui, p é branco. Pinte-o de
azul.

6) Coloque no conjunto V dos pi-
xels a serem visitados os pixels
à direita, acima, à esquerda e
abaixo de p.


(159,165) são as coordenadas da
semente.
```

Algoritmo 4: Crescimento de semente em C++/OpenCV usando fila.

Para compilar algoritmo 4 com OpenCV:

```
nocek$ compila.sh pinta_nocek
cek$   compila pinta_nocek -ocv
```

O cabeçalho “queue” (linha 3) declara as classes e funções que manipulam filas.

O programa começa na linha 24. Lê o arquivo “mickey_reduz.bmp” e o armazena como imagem colorida na matriz *a*. Depois, chama a função *pintaAzul* para pintar o componente conexo da imagem *a* conectado ao pixel-semente (159, 165) de azul e armazenar o resultado em *b*. Por fim, grava *b* como “fila.png”.

O desafio real é escrever a função *pintaAzul*. Na linha 7, o programa cria uma cópia da imagem *a* usando o método *clone()* e a armazena em *b*. Se não usasse *clone()* (isto é se fizéssemos *b=a*), teríamos uma única matriz na memória com dois nomes diferentes (*a* e *b*).

Na linha 8, o programa cria uma fila de números inteiros *q*. Vamos inserir as coordenadas de um pixel sempre na ordem *l* seguida por *c*. Na hora de retirar, teremos que retirar na mesma ordem: *l* seguida por *c*.

Na linha 9, as coordenadas do pixel-semente são inseridas na fila *q*, com o método *q.push(ls)*.

Depois, repetimos os comandos entre as linhas 10 e 20, até que não tenha mais nenhum pixel a ser visitado, isto é, até que a fila *q* esteja vazia.

Em C++, não existe um método para retirar um elemento da fila e armazená-lo numa variável. Para isso, deve-se dar dois comandos: *l=q.front()* que olha quem está no início da fila e o armazena em *l*; e *q.pop()* para retirar o elemento que está no início da fila.

A linha 13 verifica se o elemento retirado é branco. Se for, pinta-o de azul e armazena as coordenadas dos seus 4 vizinhos na fila. Caso contrário, não faz nada e processa o próximo elemento da fila.

Na linha 21, a função *pintaAzul* devolve a imagem *b* com o componente conexo pintado de azul.

O algoritmo 4 em Python usando fila fica:

```
 1 # pintaaz2.py - 2024
 2 import cv2
 3 import queue
 4
 5 def pintaAzul(a,li,ci):
 6   b=a.copy()
 7   q=queue.Queue()
 8   q.put(li) #1
 9   q.put(ci) #1
10   while not q.empty(): #2
11     l=q.get() #3
12     c=q.get() #3
13     if (b[l,c]==(255,255,255)).all(): #4
14       b[l,c]=(255,0,0) #5
15       q.put(l-1); q.put(c) #6-acima
16       q.put(l+1); q.put(c) #6-abaixo
17       q.put(l); q.put(c+1) #6-direita
18       q.put(l); q.put(c-1) #6-esquerda
19   return b;
20
21 a = cv2.imread('mickey_reduz.bmp',
22                 cv2.IMREAD_COLOR)
23 b = pintaAzul(a,159,165)
24 cv2.imwrite('pintaaz_py.png',b)
```

```
Algoritmo 2



1) Coloque o pixel-semente no conjunto V
dos pixels a serem visitados.

2) Se o conjunto V for vazio, termine o
programa.

3) Retire um pixel p de V.

4) Se p for preto ou azul, vá para (2).

5) Aqui, p é branco. Pinte-o de azul.

6) Coloque no conjunto V dos pixels a serem
visitados os pixels à direita, acima, à es-
querda e abaixo de p.




(159,165) são as coordenadas da semente.
```

Algoritmo 5: Crescimento de semente em Python usando fila.

Para rodar este programa Python:

```
Windows> python pintaaz2.py
Linux$ python3 pintaaz2.py
```

A versão do algoritmo em Python é praticamente idêntica à C++.

Em Python, o método *l=q.get()* (linha 11) retira o elemento no início da fila *q* e o armazena em *l*.

Não tem como a função de leitura *imread* (linha 22) saber se está querendo ler imagem em níveis de cinza ou colorida, pois a variável *a* não está definida e portanto não se conhece o tipo de matriz. Assim, é necessário escrever *cv2.IMREAD_COLOR* (ou o número 1) para dizer que queremos ler o arquivo como imagem colorida.

Nota: Como sabemos, C++ é consideravelmente mais rápido do que Python. A versão C++ deste programa demorou 0.035s enquanto a versão Python demorou 0.716s (C++ foi 20× mais rápido que Python). Existem bibliotecas (como Numba) que aceleram o código Python. Mas muitas vezes (como no Algoritmo 4) requer que o código Python seja reescrito para que possa ser acelerado por Numba.

Exercício: Tanto o algoritmo 4 como o algoritmo 5 podem dar erro se a área a ser pintada estiver grudada à borda da imagem, como na figura 7. Modifique os algoritmos para que os programas funcionem mesmo neste caso.

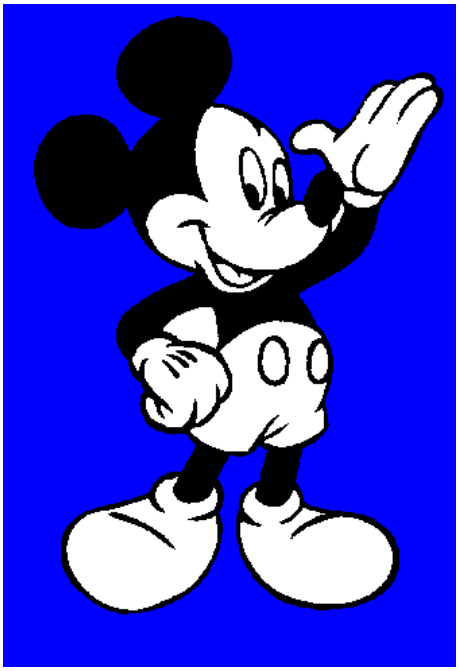


Figura 7: Os algoritmos acima não conseguem pintar uma área grudada à borda da imagem.

5. Pilha

A pilha é uma estrutura do tipo last-in-first-out (LIFO). É como uma pilha de livros: o último livro colocado na pilha vai para o topo da pilha. Na hora de retirar, quem está no topo da pilha é retirado primeiro.

início do programa	pilha vazia
Inserir A na pilha.	(topo da pilha) A (fundo da pilha)
Inserir B na pilha.	(topo da pilha) BA (fundo da pilha)
Retirar um elemento da pilha: Sai B.	(topo da pilha) A (fundo da pilha)
Inserir C na pilha.	(topo da pilha) CA (fundo da pilha)
Retirar um elemento da pilha: Sai C.	(topo da pilha) A (fundo da pilha)
Retirar um elemento da pilha: Sai A.	pilha vazia

Figura 8: Operação de pilha.

Exercício: Deixo para o aluno a tarefa de reescrever os algoritmos 3 e 4 usando pilha (em vez de fila). O cuidado que se deve tomar é a ordem em que os números são inseridos e retirados. Se inserir as coordenadas de um pixel na ordem *l* seguida por *c* usando pilha, teremos que retirar *c* primeiro e depois *l*. Pense por quê.

Atividade: Assistam os vídeos:
http://www.lps.usp.br/hae/apostila/pintafila.mp4
http://www.lps.usp.br/hae/apostila/pintapilha.mp4
para ver como o uso de fila e pilha muda a ordem em que os pixels são pintados. Repare que, usando fila, aqueles pixels que estão mais próximos da semente (usando distância 4-conexa geodésica, isto é, o menor caminho 4-conexo sem passar pelas paredes) são pintados primeiro. Usando pilha, enquanto o algoritmo puder pintar o pixel de cima, o algoritmo para caminhar para cima. Só quando não dá mais para ir para cima, o algoritmo vai procurar outras direções possíveis.

[PSI3471 aula 2 parte 2. Pular a partir daqui <<<<<<<<<<<<<<<<<<<<<<]

6. Recursão

Recursão é chamar uma função *f* dentro da própria função. Quando uma função *f* é chamada, o programa armazena: (a) o endereço para onde deve retornar quando terminar a execução da função *f*; (b) os parâmetros com que a função *f* foi chamada. Estes dados são armazenados numa pilha. Portanto, recursão cria implicitamente uma pilha, que podemos usar para implementar o crescimento de semente.

```
 1 //recursao.cpp - grad2018
 2 #include <cekeikon.h>
 3
 4 void pintaAzul(Mat_<COR>& a, int l, int c)
 5 { if (a(l,c)==COR(255,255,255)) {
 6     a(l,c)=COR(255,0,0);
 7     pintaAzul(a,l-1,c);
 8     pintaAzul(a,l,c-1);
 9     pintaAzul(a,l+1,c);
10     pintaAzul(a,l,c+1);
11   }
12 }
13
14 int main() {
15   Mat_<COR> a;
16   le(a,"mickey_reduz.bmp");
17   pintaAzul(a,159,165);
18   imp(a,"recursao.png");
19 }
```

Algoritmo 5: Crescimento de semente em C++ usando recursão.

O algoritmo 5, recursivo, é consideravelmente mais simples do que as versões com fila ou pilha, pois não há declaração explícita de uma estrutura de dados para armazenar os pixels que se deve visitar. No programa, há 5 chamadas da função "pintaAzul" (linhas 8, 9, 10, 11, e 18). O programa deve guardar a linha (na verdade, o endereço) do programa que chamou a função "pintaAzul" pois, caso contrário, não há como saber para qual das 5 linhas deve voltar quando terminar a execução da função *pintaAzul*. Além disso, os parâmetros *a*, *l* e *c* também são armazenados na pilha a cada chamada da função *pintaAzul*, para que cada instância da função trabalhe com os valores dos parâmetros com que foi chamado.

A imagem *a* é passada por referência (operador & da linha 4) para minimizar os dados armazenados na pilha. Assim, será armazenado o endereço de *a* (64 bits) na pilha e não o header de *a*.

Nota: Em Linux, não precisa fazer nada de especial para compilar e executar o programa acima. Em Windows, o compilador GCC do MinGW (usado no Cekeikon) reserva um espaço limitado para pilha (stack). Assim, se mandar pintar uma área grande, a memória stack irá acabar e o programa irá travar, sem que o programa emita qualquer mensagem de erro. Para evitar isso, usando "compila" do Cekeikon, escreva:

```
>compila recursao -cek -stack=5000000
```

A cada chamada recursiva, gastam-se 64+64+32+32=192 bits=24 bytes (em compilador de 64 bits). A imagem tem aproximadamente 166.000 pixels. Como pode haver 4 chamadas de recursão para cada pixel, tamanho da pilha deve ser maior que 5 MBytes para poder rodar sem preocupação. [Verificar se a conta está correta. Talvez o certo seja 15.936.000 bytes].

Se for chamar diretamente o compilador GCC do MinGW-Windows, especifique a opcao:

"-Wl,-stack=5000000"

no compilador GCC, onde 5000000 (ou maior) é o tamanho do stack em bytes. É necessário usar realmente os colchetes na chamada do compilador.

Exercício: Escreva um programa que lê uma imagem binária e imprime o número de componentes conexos da imagem. Rodando o seu programa para imagem "letras.bmp", deve retornar 31. Aqui, estou chamando de componente conexo o conjunto de pixels pretos grudados entre si. Execute o programa também para imagens c2.bmp e c3.bmp.

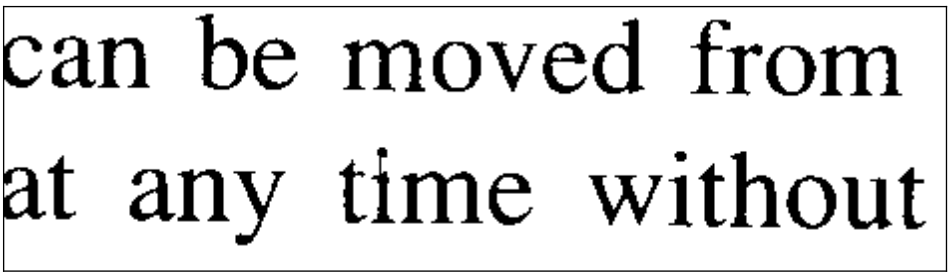
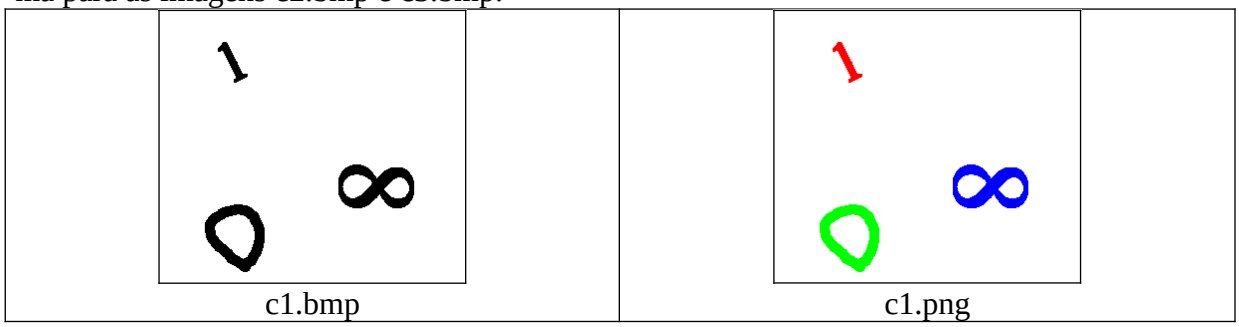


Figura 9: Letras.bmp contém 31 componentes conexos. Numa das letras, o pingo do *i* foi grudado propositalmente ao corpo do caracter.

[PSI3471 aula 2 parte 2. Pular até aqui <<<<<<<<<<<<<<<<<<<<<<]

[PSI3471 aula 2. Lição de casa extra #2. Vale +2 pontos.] Escreva um programa que, dada uma imagem binária, pinta de vermelho os componentes conexos sem furo, de verde os componentes com um furo e de azul os componentes com dois ou mais furos. Rode o seu programa para as imagens c2.bmp e c3.bmp.



c1.bmp c1.png

Figura 10: Pintar cada componente conexo de acordo com o número de furos.

Exercício: Escreva um programa que, dada uma imagem binária, elimina todos os componentes conexos com menos de *n* pixels, onde *n* é um parâmetro especificado pelo usuário. Este programa pode ser usada para eliminar ruídos em imagens binárias.

Exercício: Escreva um programa que, dada uma imagem colorida, as coordenadas (l_s, c_s) do pixel-semente *s* e um parâmetro de tolerância *t*, pinta de vermelho todos os pixels *p* conectados ao pixel-semente *s* cuja distância euclideana no espaço das cores RGB seja menor que a tolerância *t*, isto é, distância*(p, s)<t.*

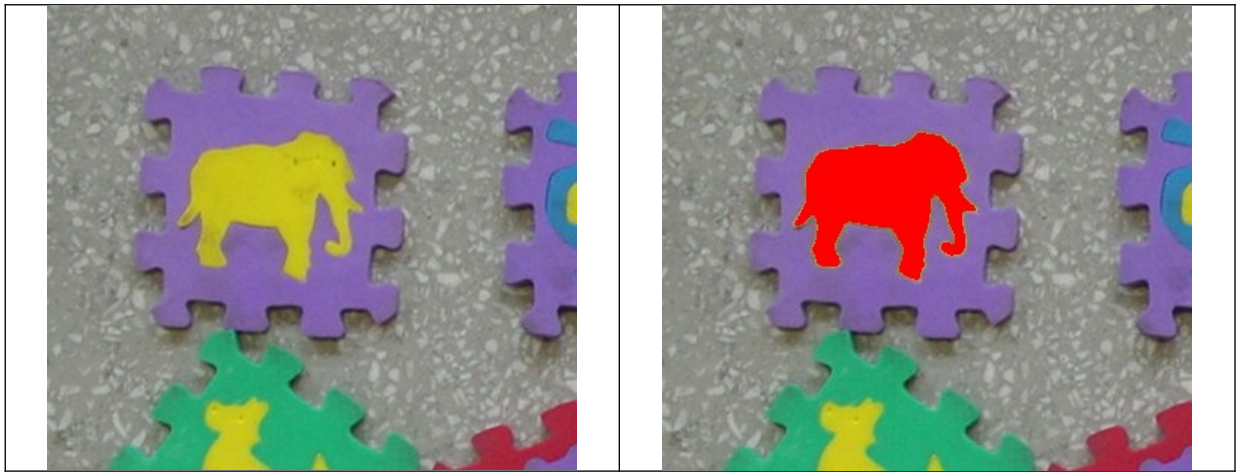


Figura 11: Crescimento de semente numa imagem colorida.

[PSI3471 aula 2 parte 2. Fim]

7. O menor caminho entre rato e queijo.

Algoritmo de Dijkstra [Dijkstra1959, WikiDijkstra] encontra o menor caminho entre dois nós de um grafo. Vamos adaptar este algoritmo para encontrar o menor caminho entre dois pixels (por exemplo, entre rato e queijo) de um labirinto, usando fila.

O nosso problema é, dado um labirinto com rato *r* no canto superior esquerdo e o queijo *q* no canto inferior direito, achar o menor caminho do rato para o queijo (sem passar pelas paredes do labirinto e considerando 4-conectividade, figura 10). É possível resolver este problema adaptando o algoritmo de crescimento de semente.

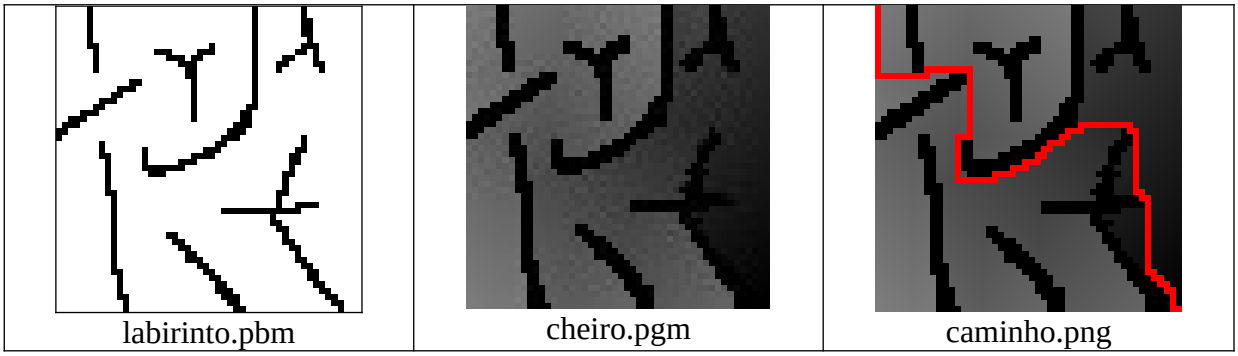


Figura 10: Achar o menor caminho entre rato (canto superior esquerdo) e queijo (canto inferior direito).

Alguma ideia de como se pode resolver este problema? A ideia intuitiva é fazer queijo "emitir cheiro", isto é vamos criar uma matriz "cheiro" que indica, para cada pixel *p* da matriz, a menor distância geodésica (isto é, distância sem passar pelas paredes e podendo andar só nas 4 direções: norte, sul, leste, oeste) entre *p* e *q*. Por exemplo, se o rato estiver num pixel *p* e precisa dar pelo menos 4 passos para chegar até o queijo *q*, então *cheiro(p)=4*. Quando a matriz *cheiro* estiver construída, basta fazer o rato andar, a partir da sua posição inicial, sempre para o pixel vizinho onde o cheiro do queijo fica mais forte (ou, equivalentemente, para o pixel vizinho que deixa o rato mais perto do queijo). Este processo está ilustrado na figura 10. O algoritmo está dividido em duas partes: *espalhaCheiro* que constrói a matriz *cheiro* e *tracaCaminho* que traça o caminho mais curto do rato até o queijo.

7.1 EspalhaCheiro

É possível escrever a função *espalhaCheiro* (algoritmo 6) modificando o crescimento de semente com fila (*pintafila*, algoritmo 3). Não se pode usar pilha ou recursão. Assista novamente os vídeos:

http://www.lps.usp.br/hae/apostila/pintafila.mp4
http://www.lps.usp.br/hae/apostila/pintapilha.mp4

Repare que *pintafila* sempre pinta primeiro aqueles pixels que estão mais próximos do pixel semente. Podemos usar esta propriedade para construir a matriz *cheiro*. O programa *pintapilha* não possui esta propriedade e portanto não podemos usá-lo neste problema.

```
 1 //espalhacheiro.cpp pos-2020
 2 #include <cekeikon.h>
 3 #include <queue>
 4
 5 void espalhaCheiro(Mat_<GRY> che, int ls, int cs) {
 6   queue<int> q;
 7   q.push(ls); q.push(cs); q.push(0);
 8   while (!q.empty()) {
 9     int l=q.front(); q.pop();
10     int c=q.front(); q.pop();
11     int d=q.front(); q.pop();
12     if (0<=l && l<che.rows && 0<=c && c<che.cols && che(l,c)==255) {
13       che(l,c)=d;
14       q.push(l-1); q.push(c); q.push(d+1);
15       q.push(l+1); q.push(c); q.push(d+1);
16       q.push(l); q.push(c+1); q.push(d+1);
17       q.push(l); q.push(c-1); q.push(d+1);
18     }
19   }
20 }
21
22 int main() {
23   Mat_<GRY> che; le(che,"labirinto.pbm");
24   espalhaCheiro(che,che.rows-1,che.cols-1);
25   imp(che,"cheiro.pgm");
26 }
```

Algoritmo 6: "Espalha cheiro".

O labirinto inicialmente possui valores 0 nas paredes e valores 255 nos pixels livres onde o rato pode andar. À medida que o programa é executado, os pixels livres do labirinto vão sendo pintados com a distância geodésica até o queijo. Assista o vídeo:

http://www.lps.usp.br/hae/apostila/espalhacheiro.mp4

que mostra como o cheiro se espalha a partir do queijo colocado no centro da imagem.

O algoritmo *pintafila* só precisa pintar de azul os pixels conectados à semente. Assim, armazena na fila apenas as coordenadas (l, c) dos pixels a serem visitados. O algoritmo *espalhaCheiro* precisa pintar cada pixel p conectado à semente (isto é, o queijo) com a distância geodésica do pixel p até o queijo q. Para isso, precisa armazenar na fila, além das coordenadas, a distância geodésica d do pixel p até q. Esta distância é sempre um a mais do que a distância do pixel vizinho de p de onde chegou o cheiro do queijo.

Na linha 7, o programa armazena as coordenadas l_s e c_s da semente e a distância 0 (pois a distância do queijo até queijo é zero). Nas linhas 14-17, armazena d+1 na fila, pois a distância geodésica de um pixel p até q é sempre um a mais que a distância d do pixel vizinho de p de onde chegou o cheiro do queijo.

Na linha 12, além de testar se o pixel (l, c) tem valor 255 (pixel livre), também verifica se (l, c) pertence ao domínio da imagem, para evitar pintar os pixels fora do domínio. Se o teste falhar, joga fora o pixel (l, c) e faz teste com o próximo pixel da fila.

Na linha 13, o pixel atual (l, c) é um pixel livre, pois passou pelo teste da linha 12. Pinta-o com a distância d retirado da fila.

Nas linhas 14-17 coloca os quatro vizinhos do pixel atual (l, c) na fila, juntamente com a distância geodésica d+1.

Este processo termina se não houver mais pixels a serem visitados (linha 7).

Nota: Usei Mat_<GRY> para armazenar matriz *cheiro*, pois isso permite visualizar essa matriz como uma imagem em níveis de cinzas. Por outro lado, como cada pixel de Mat_<GRY> só consegue armazenar números de 0 a 255, se a distância geodésica de algum pixel for maior que 254, o programa não funcionará corretamente (usamos 255 para representar os pixels livres do labirinto). Neste caso, devemos utilizar Mat_<short>, Mat_<float>, MAT_<int> ou qualquer outra estrutura matricial que consiga armazenar números grandes.

7.1 TracaCaminho

Após criar a matriz *cheiro*, devemos traçar o menor caminho do rato até o queijo. Na verdade, é possível traçar o menor caminho de qualquer ponto (l_r, c_r) do labirinto até o queijo, se esse ponto estiver conectado ao queijo. Estamos supondo que queijo fica no canto inferior direito.

```
 1 //tracacaminho.cpp pos-2020
 2 #include <cekeikon.h>
 3
 4 void tracaCaminho(Mat_<COR> cam, int lr, int cr) {
 5   int d=cam(lr,cr)[0];
 6   if (d==255) return; //Nao ha caminho entre (lr,cr) e queijo
 7   while (d>0) {
 8     cam(lr,cr)=COR(0,0,255);
 9     d=d-1;
10     if      (cam(lr-1,cr)[0]==d) lr=lr-1;
11     else if (cam(lr+1,cr)[0]==d) lr=lr+1;
12     else if (cam(lr,cr-1)[0]==d) cr=cr-1;
13     else if (cam(lr,cr+1)[0]==d) cr=cr+1;
14   }
15   cam(cam.rows-1,cam.cols-1)=COR(0,0,255);
16 }
17
18 int main() {
19   Mat_<COR> cam; le(cam,"cheiro.pgm");
20   tracaCaminho(cam,0,0);
21   imp(cam,"caminho.png");
22 }
```

O programa acima lê *cheiro.pgm* gerado pelo programa *espalhaCheiro* como uma imagem colorida (linha 19). Os componentes B, G e R de todos os pixels de matriz *cam* são iguais, pois uma imagem em níveis de cinza foi lida como colorida.

Na linha 6, o programa testa se o pixel (l_r, c_r) está conectado ao queijo. Se o valor do pixel for 255 significa que o cheiro não chegou até (l_r, c_r) e este pixel não está conectado ao queijo.

Nas linhas 9-13 procura, entre os pixels vizinhos, aquele onde o cheiro do queijo é mais forte (ou diminui a distância geodésica). O rato deve andar para esse pixel.

Referências:

[Rosenfeld1966] ROSENFELD, A.; PFALTZ, J. L. Sequential operations in digital picture processing. Journal of the ACM, v. 13, n. 4, p. 471–494, out. 1966.

[Santiago2019] Diêgo J. C. Santiago, Algoritmos rápidos para rotulação de componentes conexos utilizando blocos 2x2, tese de doutorado, UFPE, 2019.

[WikiGrowing] https://en.wikipedia.org/wiki/Region_growing, acessado em 20 de março de 2020.

[Dijkstra1959] Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". *Numerische Mathematik.* ***1****: 269–271.*

[WikiDijkstra] *https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm*, acessado em 20 de março de 2020.