CAPÍTULO

5

COLORAÇÃO

Mostraremos agora uma interessante aplicação prática para a estrutura de dados fila, implementada no Capítulo 4, que ilustra conceitos usados em computação gráfica.

5.1 Fundamentos

Nesse capítulo, usamos o tipo `Fila` para criar um programa que colore regiões de uma imagem com uma nova cor. Essa operação é comum em programas de desenho interativo, como ilustrado na Figura 5.1. O usuário ativa a operação de coloração, seleciona a cor desejada e, ao clicar em um ponto interno a uma região da imagem, o programa colore essa região com a nova cor selecionada.

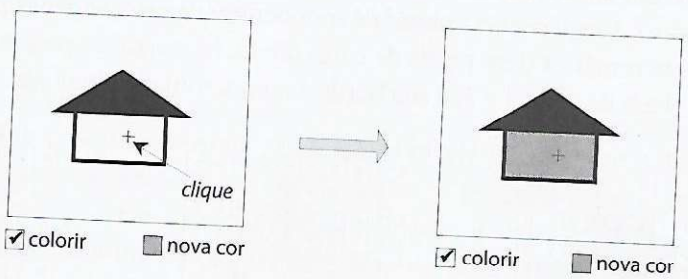
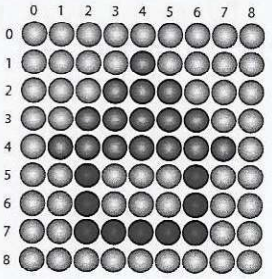


Figura 5.1 | Coloração de imagem.

5.1.1 Formação de imagem

Uma *imagem* é formada por uma matriz bidimensional de pontos luminosos no vídeo. Cada um desses pontos, chamado *pixel* (*picture element*), é representado na memória por um número, de acordo com a sua cor na imagem. A quantidade de cores depende da capacidade do vídeo usado para exibir a imagem.

Por exemplo, a Figura 5.2 ilustra a imagem para o desenho da Figura 5.1 e sua possível representação na memória.



(a) imagem no vídeo

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0
2	0	0	0	1	1	1	0	0	0
3	0	0	1	1	1	1	1	0	0
4	0	1	1	1	1	1	1	1	0
5	0	0	2	0	0	0	2	0	0
6	0	0	2	0	0	0	2	0	0
7	0	0	2	2	2	2	2	0	0
8	0	0	0	0	0	0	0	0	0

(b) representação na memória

Figura 5.2 | Uma imagem formada por pixels e sua representação na memória.

5.1.2 Algoritmos de coloração de imagem

Há dois tipos de algoritmos de coloração de imagem:

- Coloração *limitada por área* (*flood-fill*) considera regiões monocromáticas limitadas por bordas policromáticas. Dado um pixel p de cor a, uma nova cor n é propagada para todo pixel de cor a na região de p.
- Coloração *limitada por borda* (*boundary-fill*) considera regiões policromáticas limitadas por bordas monocromáticas. Dado um pixel p e uma cor de borda b, uma nova cor n é propagada para todo pixel de cor distinta de n e de b na região de p.

A Figura 5.3 mostra a diferença entre os resultados obtidos com esses dois tipos de algoritmos. Em ambos os casos, a coloração com a nova cor 3 é iniciada no pixel (2, 4). No primeiro caso, a região a ser colorida é monocromática (tem apenas pixels de cor 1) e sua borda é policromática (tem pixels de cores 0 e 2). No segundo caso, a região é policromática (tem pixels de cores 1 e 2) e sua borda é monocromática (tem apenas pixels de cor 0).

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0
2	0	0	0	1	1	1	0	0	0
3	0	0	1	1	1	1	1	0	0
4	0	1	1	1	1	1	1	1	0
5	0	0	2	0	0	0	2	0	0
6	0	0	2	0	0	0	2	0	0
7	0	0	2	2	2	2	2	0	0
8	0	0	0	0	0	0	0	0	0

(a) imagem original

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	3	0	0	0	0
2	0	0	0	3	3	3	0	0	0
3	0	0	3	3	3	3	3	0	0
4	0	3	3	3	3	3	3	3	0
5	0	0	2	0	0	0	2	0	0
6	0	0	2	0	0	0	2	0	0
7	0	0	2	2	2	2	2	0	0
8	0	0	0	0	0	0	0	0	0

(b) limitado por área de cor 1

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	3	0	0	0	0
2	0	0	0	3	3	3	0	0	0
3	0	0	3	3	3	3	3	0	0
4	0	3	3	3	3	3	3	3	0
5	0	0	3	0	0	0	3	0	0
6	0	0	3	0	0	0	3	0	0
7	0	0	3	3	3	3	3	0	0
8	0	0	0	0	0	0	0	0	0

(c) limitado por borda de cor 0

Figura 5.3 | Diferença entre coloração limitada por área e por borda.

A coloração limitada por área é mais flexível que aquela limitada por borda, pois o resultado da última pode ser obtido pela sucessiva aplicação da primeira. Por exemplo, aplicando a coloração limitada por borda na Figura 5.3a, em um passo, obtemos a Figura 5.3c; mas isso também pode ser feito com a coloração limitada por área em dois passos, por exemplo: primeiro, colorimos a área do pixel (2, 4) na Figura 5.3a e obtemos a Figura 5.3b; em seguida, colorimos a área do pixel (5, 2) na Figura 5.3b e obtemos a Figura 5.3c. Por esse motivo, no programa que será criado, usaremos a coloração limitada por área.

5.1.3 Coloração limitada por área

Os *vizinhos* de um pixel em uma imagem são aqueles que se encontram acima, à direita, abaixo ou à esquerda dele, como ilustrado na Figura 5.4.

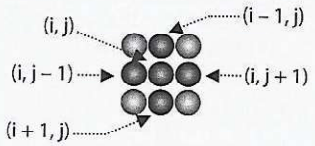


Figura 5.4 | Vizinhança de um pixel numa posição (i,j) de uma imagem.

Sejam `p` e `q` dois pixels de mesma cor. Então, `q` está na *região* de `p` se ele é vizinho de `p` ou de algum outro pixel que está na região de `p`. Com base nessa definição, dados um pixel `p` e uma nova cor `n`, a coloração da região de `p` é feita do seguinte modo:

- Crie uma fila vazia `F`.
- Obtenha cor atual `a` de `p`.
- Mude a cor de `p` para `n`.
- Enfileire `p` em `F`.
- Enquanto a fila `F` não estiver vazia, faça:
  - Desenfileire um pixel `p` de `F`.
  - Para cada vizinho `q` de `p`, que tenha a cor `a`, faça:
    - Mude a cor de `q` para `n`.
    - Enfileire `q` em `F`.

A finalidade da fila `F` nesse processo é manter os pixels já coloridos, até que seus vizinhos possam ser inspecionados e, eventualmente, também coloridos. Assim, quando `F` fica vazia, temos certeza de que todos os pixels de cor `a`, acessíveis a partir de `p`, foram coloridos com a nova cor `n`. O processo de coloração da região do pixel `(5,4)`, com uma nova cor `4`, é ilustrado na Figura 5.5.

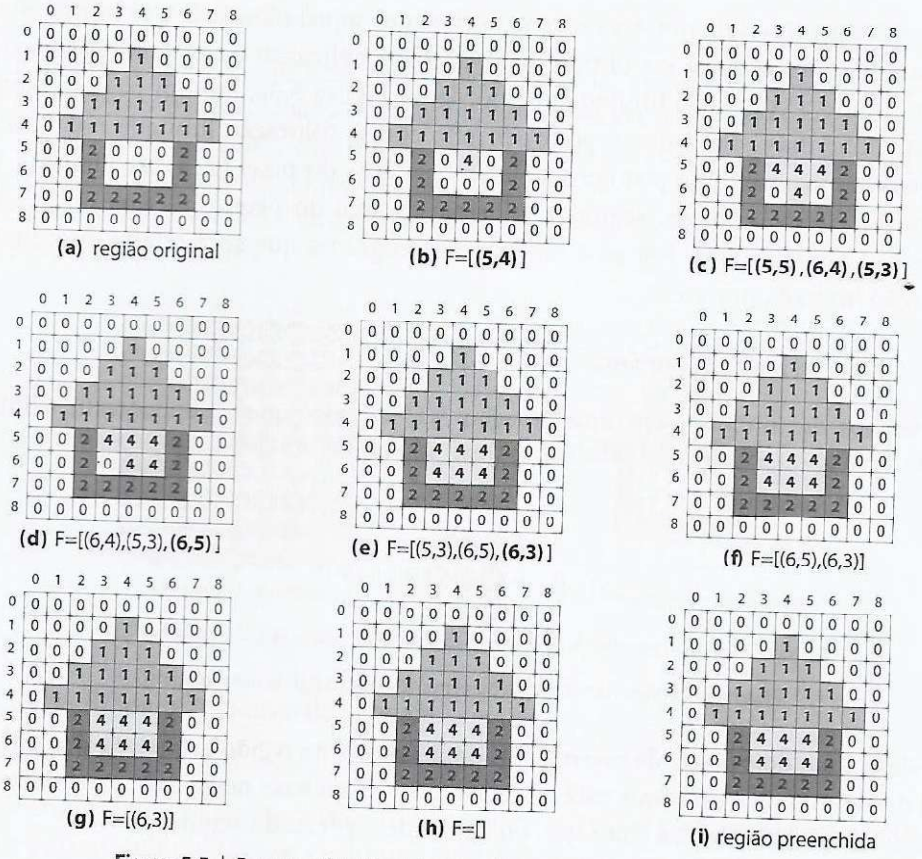


Figura 5.5 | Processo de coloração da região do pixel (5,4) com a nova cor 4.

5.2 Representação e exibição de imagem

A imagem a ser manipulada pelo algoritmo de coloração de regiões será representada por uma matriz de números inteiros positivos. Por exemplo, a imagem da Figura 5.2 será representada pela matriz definida na Figura 5.6.

```
#define dim 9      // dimensao da matriz
int I[dim][dim] = {{0, 0, 0, 0, 0, 0, 0, 0, 0},
                   {0, 0, 0, 0, 1, 0, 0, 0, 0},
                   {0, 0, 0, 1, 1, 1, 0, 0, 0},
                   {0, 0, 1, 1, 1, 1, 1, 0, 0},
                   {0, 1, 1, 1, 1, 1, 1, 1, 0},
                   {0, 0, 2, 0, 0, 0, 2, 0, 0},
                   {0, 0, 2, 0, 0, 0, 2, 0, 0},
                   {0, 0, 2, 2, 2, 2, 2, 0, 0},
                   {0, 0, 0, 0, 0, 0, 0, 0, 0}};
```

Figura 5.6 | Representação de uma imagem como uma matriz quadrada em C.

A função `exiba()`, na Figura 5.7, exibe no vídeo a imagem representada pela matriz `I`. Como C padrão não tem recursos para exibição em modo gráfico, essa função usa o caractere '█' (ASCII 219) para representar os pixels da imagem no vídeo. Porém, como a altura desse caractere é o dobro de sua largura, para manter a proporção, cada pixel é exibido como um par de caracteres. A função `_textcolor()`, declarada em `conio.h`, é usada para ativar a cor de exibição dos caracteres que representam o pixel `I[i][j]` no vídeo. Essa função requer como parâmetro um número entre `0` e `15`, conforme indicado na Figura 5.8.

```
void exiba(int I[dim][dim]) {
   for(int i=-1; i<dim; i++) {
      _textcolor(8);
      for(int j=-1; j<dim; j++)
         if( i<0 && j<0 ) printf("  ");
         else if( i<0 ) printf("%2d",j);
         else if( j<0 ) printf("\n%2d",i);
         else { _textcolor(I[i][j]); printf("%c%c",219,219); }
   }
   _textcolor(8);
}
```

Figura 5.7 | Função para exibição de matriz que representa uma imagem.

Cor	Número	RGB	Cor	Número	RGB
preto	0	0000	cinza escuro	8	1000
azul	1	0001	azul claro	9	1001
verde	2	0010	verde claro	10	1010
ciano	3	0011	ciano claro	11	1011
vermelho	4	0100	vermelho claro	12	1100
magenta	5	0101	magenta claro	13	1101
amarelo	6	0110	amarelo claro	14	1110
cinza claro	7	0111	branco	15	1111

Figura 5.8 | Cores usadas pela função `_textcolor()`.

Na função `exiba()`, as variáveis `i` e `j` são iniciadas com o valor `-1`. Desse modo, além de exibir a imagem, a função pode exibir também as coordenadas de cada pixel da imagem. Por exemplo, para a matriz `I` definida na Figura 5.6, a chamada `exiba(I)` produz uma saída similar àquela apresentada na Figura 5.9.

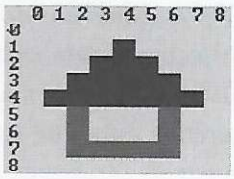


Figura 5.9 | Saída produzida pela função exiba().

5.3 Coloração de imagem

Nessa seção, apresentamos a implementação do algoritmo de coloração limitada por área e discutimos alguns detalhes dessa implementação.

5.3.1 Vizinhança nas margens da imagem

A definição de vizinhança apresentada anteriormente supõe que todo pixel em uma imagem tem quatro vizinhos (Figura 5.4). Entretanto, essa suposição não vale para os pixels nas margens da imagem, que podem ter apenas dois ou três vizinhos. Por exemplo, o pixel `(0,0)` numa imagem não tem vizinho acima nem à esquerda. Assim, ao inspecionar os vizinhos de um pixel numa imagem, há o risco de acessar posições inválidas na matriz que representa essa imagem. Para evitar esse risco, usaremos a macro definida na Figura 5.10.

```
#define cor(i,j) (i>=0  && i<dim && j>=0 && j<dim ? I[i][j] : -1)
```

Figura 5.10 | Macro para tratamento de pixels na margem da imagem.

Caso a macro seja chamada com uma posição válida, o resultado é a cor do pixel nessa posição; caso contrário, o resultado é `-1`. Como não há cor negativa, `-1` indica ausência de pixel, evitando acessos a posições inválidas na matriz.

5.3.2 Manipulação de coordenadas

Como a posição de um pixel numa imagem é um par `(i,j)`, para implementar o algoritmo de coloração, podemos usar duas filas de inteiros: uma para guardar valores de `i` e outra para guardar valores de `j`. Entretanto, para tornar o código mais conciso, usaremos uma única fila e guardaremos cada par `(i,j)` como um único número inteiro da forma `i*100+j`.

```
#define par(i,j) ((i)*100+(j))
#define lin(p)   ((p)/100)
#define col(p)   ((p)%100)
```

Figura 5.11 | Macros para manipulação de coordenadas.

A função `exiba()`, na Figura 5.7, exibe no vídeo a imagem representada pela matriz `I`. Como C padrão não tem recursos para exibição em modo gráfico, essa função usa o caractere '▮' (ASCII 219) para representar os pixels da imagem no vídeo. Porém, como a altura desse caractere é o dobro de sua largura, para manter a proporção, cada pixel é exibido como um par de caracteres. A função `_textcolor()`, declarada em `conio.h`, é usada para ativar a cor de exibição dos caracteres que representam o pixel `I[i][j]` no vídeo. Essa função requer como parâmetro um número entre `0` e `15`, conforme indicado na Figura 5.8.

```
void exiba(int I[dim][dim]) {
   for(int i=-1; i<dim; i++) {
      _textcolor(8);
      for(int j=-1; j<dim; j++)
         if( i<0 && j<0 ) printf("  ");
         else if( i<0 ) printf("%2d",j);
         else if( j<0 ) printf("\n%2d",i);
         else { _textcolor(I[i][j]); printf("%c%c",219,219); }
   }
   _textcolor(8);
}
```

Figura 5.7 | Função para exibição de matriz que representa uma imagem.

Cor	Número	RGB	Cor	Número	RGB
preto	0	0000	cinza escuro	8	1000
azul	1	0001	azul claro	9	1001
verde	2	0010	verde claro	10	1010
ciano	3	0011	ciano claro	11	1011
vermelho	4	0100	vermelho claro	12	1100
magenta	5	0101	magenta claro	13	1101
amarelo	6	0110	amarelo claro	14	1110
cinza claro	7	0111	branco	15	1111

Figura 5.8 | Cores usadas pela função `_textcolor()`.

Na função `exiba()`, as variáveis `i` e `j` são iniciadas com o valor `-1`. Desse modo, além de exibir a imagem, a função pode exibir também as coordenadas de cada pixel da imagem. Por exemplo, para a matriz `I` definida na Figura 5.6, a chamada `exiba(I)` produz uma saída similar àquela apresentada na Figura 5.9.

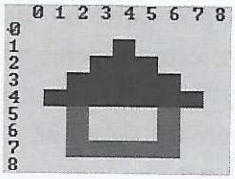


Figura 5.9 | Saída produzida pela função exiba().

5.3 Coloração de imagem

Nessa seção, apresentamos a implementação do algoritmo de coloração limitada por área e discutimos alguns detalhes dessa implementação.

5.3.1 Vizinhança nas margens da imagem

A definição de vizinhança apresentada anteriormente supõe que todo pixel em uma imagem tem quatro vizinhos (Figura 5.4). Entretanto, essa suposição não vale para os pixels nas margens da imagem, que podem ter apenas dois ou três vizinhos. Por exemplo, o pixel `(0,0)` numa imagem não tem vizinho acima nem à esquerda. Assim, ao inspecionar os vizinhos de um pixel numa imagem, há o risco de acessar posições inválidas na matriz que representa essa imagem. Para evitar esse risco, usaremos a macro definida na Figura 5.10.

```
#define cor(i,j) (i>=0  && i<dim && j>=0 && j<dim ? I[i][j] : -1)
```

Figura 5.10 | Macro para tratamento de pixels na margem da imagem.

Caso a macro seja chamada com uma posição válida, o resultado é a cor do pixel nessa posição; caso contrário, o resultado é `-1`. Como não há cor negativa, `-1` indica ausência de pixel, evitando acessos a posições inválidas na matriz.

5.3.2 Manipulação de coordenadas

Como a posição de um pixel numa imagem é um par `(i,j)`, para implementar o algoritmo de coloração, podemos usar duas filas de inteiros: uma para guardar valores de `i` e outra para guardar valores de `j`. Entretanto, para tornar o código mais conciso, usaremos uma única fila e guardaremos cada par `(i,j)` como um único número inteiro da forma `i*100+j`.

```
#define par(i,j) ((i)*100+(j))
#define lin(p)   ((p)/100)
#define col(p)   ((p)%100)
```

Figura 5.11 | Macros para manipulação de coordenadas.

A transformação de um par num único número correspondente é feita por `par()`, definida na Figura 5.11. Por exemplo, a chamada `par(5,19)` resulta no número `519`. Esse esquema de transformação permite lidar apenas com posições cujos números de linha e coluna estejam entre `0` e `99`. Porém, como o vídeo no modo texto tem apenas `25` linhas e `80` colunas, essa restrição não é problema.

Inversamente, a transformação de um único número num par correspondente é feita por `lin()` e `col()`, definidas na Figura 5.11. Por exemplo, as chamadas `lin(519)` e `col(519)` resultam, respectivamente, na linha `5` e na coluna `19`.

5.3.3 Coloração limitada por área

A função `colorir()`, definida na Figura 5.12, implementa o algoritmo de coloração de regiões limitada por área. Quando chamada, essa função deve receber a matriz `I` representando a imagem a ser manipulada, a posição `(i,j)` do pixel em que a coloração terá início e a nova cor `n` para essa região.

```
void colorir(int I[dim][dim], int i, int j, int n) {
   Fila F = fila(dim*dim);
   int a = I[i][j];
   I[i][j] = n;
   enfileira(par(i,j),F);
   while( !vaziaf(F) ) {
      int p = desenfileira(F);
      i = lin(p);
      j = col(p);
      if( cor(i-1,j)==a ) { I[i-1][j]=n; enfileira(par(i-1,j),F); }
      if( cor(i,j+1)==a ) { I[i][j+1]=n; enfileira(par(i,j+1),F); }
      if( cor(i+1,j)==a ) { I[i+1][j]=n; enfileira(par(i+1,j),F); }
      if( cor(i,j-1)==a ) { I[i][j-1]=n; enfileira(par(i,j-1),F); }
   }
   destroif(&F);
}
```

Figura 5.12 | Função que colore a região do pixel (i,j) na imagem I, com nova cor n.

Note que a cor original de um pixel na posição `(i,j)` é sempre um número positivo. Portanto, uma condição da forma `cor(x,y)==a` será sempre falsa para uma posição `(x,y)` inválida (que tem cor `-1`), evitando que um acesso da forma `I[x][y]=n` seja feito e cause problemas na execução do programa.

5.4 O programa completo

O programa completo para coloração de imagem é apresentado na Figura 5.13. Esse programa solicita ao usuário que informe uma nova cor e a posição do pixel em que a coloração será iniciada. Para evitar que uma posição inválida seja aceita como

entrada, o programa principal repete a solicitação dessa posição até que ela esteja dentro dos limites permitidos para a imagem que será colorida. O usuário pode realizar vários experimentos com a imagem, colorindo diversas regiões com diferentes cores. Para finalizar, o usuário deve escolher uma cor inválida, representada pelo número -1.

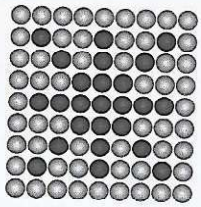
```
//coloracao.c - coloracao limitada por área

#include <stdio.h>
#include <conio.h>
#include "../ed/fila.h" // fila de int

#define dim 9
#define cor(i,j) (i>=0  && i<dim && j>=0 && j<dim ? I[i][j] : -1)
#define par(i,j) ((i)*100+(j))
#define lin(p)   ((p)/100)
#define col(p)   ((p)%100)

int I[dim][dim] = {{0, 0, 0, 0, 0, 0, 0, 0, 0},
                   {0, 0, 0, 0, 1, 0, 0, 0, 0},
                   {0, 0, 0, 1, 1, 1, 0, 0, 0},
                   {0, 0, 1, 1, 1, 1, 1, 0, 0},
                   {0, 1, 1, 1, 1, 1, 1, 1, 0},
                   {0, 0, 2, 0, 0, 0, 2, 0, 0},
                   {0, 0, 2, 0, 0, 0, 2, 0, 0},
                   {0, 0, 2, 2, 2, 2, 2, 0, 0},
                   {0, 0, 0, 0, 0, 0, 0, 0, 0}};

// adicione a função exiba(), da Figura 5.7
// adicione a função colorir(), da Figura 5.12

int main(void) {
   int i, j, n;
   while( 1 ) {
      exiba(I);
      printf("\n\nNova cor (ou -1 para sair)? ");
      scanf("%d",&n);
      if( n<0 ) break;
      do {
         printf("Posicao? ");
         scanf("%d %d",&i,&j);
      } while( i<0 || i>=dim || j<0 || j>=dim );
      colorir(I,i,j,n);
   }
   return 0;
}
```

Figura 5.13 | Programa principal para coloração de regiões limitada por área.

Exercícios

5.1 Execute o programa da Figura 5.13 e veja que ele funciona corretamente; exceto quando o usuário tenta preencher uma região com a mesma cor que ela já tem. Altere a função `colorir()` para corrigir esse problema.

5.2 Altere o programa feito no Exercício 5.1 para usar a imagem a seguir:



5.3 A função a seguir inicia uma matriz `I` com dados lidos de um arquivo de texto, cujo nome é dado pela cadeia `s`. Por exemplo, para iniciar uma matriz com os dados do arquivo `imagem.txt`, indicado ao lado, basta fazer a chamada `inicia(I,"imagem.txt")`. Usando essa função, modifique o programa da Figura 5.13 para que ele leia uma imagem de um arquivo do usuário.

```
void inicia(int I[dim][dim], char *s) {
   FILE *a = fopen(s,"rt");
   if( !a ) {
       puts("arquivo não encontrado");
       abort();
   }
   for(int i=0; i<dim; i++)
      for(int j=0; j<dim; j++)
         fscanf(a,"%d",&I[i][j]);
   fclose(a);
}
```

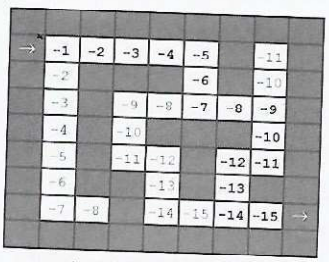
imagem.txt

```
2 2 2 2 6 2 2 2 2
2 2 2 6 6 6 2 2 2
2 2 6 6 1 6 6 2 2
2 6 6 1 1 1 6 6 2
6 6 1 1 7 1 1 6 6
2 6 6 1 1 1 6 6 2
2 2 6 6 1 6 6 2 2
2 2 2 6 6 6 2 2 2
2 2 2 2 6 2 2 2 2
```

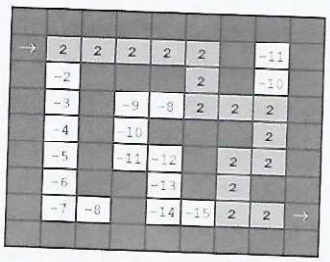
5.4 Usando coloração *limitada por área*, crie a função `inverter(I,i,j)`, que inverte a luminosidade da região do pixel `I[i][j]`. Se a região tiver uma cor clara, ela deve ser colorida com a cor escura correspondente e *vice-versa*. Conforme a Figura 5.8, a inversão da luminosidade de uma cor `c`, pode ser feita pela expressão `(c+8)%16`. Crie um programa para testar a função.

5.5 Usando coloração *limitada por borda*, crie uma versão da função `colorir()`, apresentada na Figura 5.12, e faça um programa para testá-la.

5.6 Considere um labirinto representado por uma imagem em que as cores `0` e `1` indicam *passagem* e *parede*, respectivamente, `(1,1)` a é *entrada* e `(7,7)` a é *saída*.



(a) após a coloração



(b) após a extração

Podemos adaptar a coloração *limitada por área* para encontrar um caminho mínimo da *entrada* até a *saída*, do seguinte modo: colorimos o ponto (1,1) com a cor -1 e o inserimos numa fila vazia; depois, enquanto a fila não ficar vazia, removemos um ponto (de cor n) da fila, colorimos seus vizinhos de cor 0 com a cor n-1 e enfileiramos suas posições. Após a coloração (veja a figura acima), extraímos um caminho mínimo do seguinte modo: inserimos o ponto (7,7) numa pilha vazia e, enquanto seu topo não for (1,1), empilhamos *um* vizinho do topo que tenha a cor do topo mais 1 e mudamos a cor do antigo topo para 2. Quando a repetição terminar, mudamos a cor do ponto (1,1) para 2 (então, todos os pontos do caminho mínimo extraído terão a cor 2, como indicado na figura acima). Para exibir o resultado, precisamos alterar a função de exibição, de modo que os números negativos sejam interpretados como a cor 0 (passagem). Crie um programa com base nessa ideia.