struct pixel

{

int i;

int j;

int nextDirection;

struct pixel\* next;

};

struct list

{

int i;

struct list\* next;

};

struct stack

{

int num;

struct list\* begin;

};

struct report

{

int area;

double diameter;

double perimeter;

struct pixel\* circularity;

};

void mark8(byte\*\* inputImage,

byte value,

unsigned int iseed,

unsigned int jseed,

unsigned int range\_i,

unsigned int range\_j);

void newMark8(byte\*\* inputImage,

byte value,

int iseed,

int jseed,

int range\_i,

int range\_j);

struct pixel\* chain8(byte\*\* inputImage,

byte value,

byte markValue,

unsigned int iseed,

unsigned int jseed,

unsigned int range\_i,

unsigned int range\_j);

int identifyBound(byte\*\* inputImage,

unsigned int iseed,

unsigned int jseed,

unsigned int range\_i,

unsigned int range\_j);

int range( int row,

int col,

int range\_i,

int range\_j);

#define WHITE\_LAB 255

#define BLACK\_LAB 0

int flag = 0;

int count = 0;

Image \*threshold\_lab(Image \*inputImage, unsigned int threshval){

byte \*\*image; /\* 2-d matrix data pointer \*/

unsigned int r, /\* row index \*/

c, /\* column index \*/

bands; /\* band index \*/

unsigned int no\_of\_rows, /\* number of rows in image \*/

no\_of\_cols, /\* number of columns in image \*/

no\_of\_bands; /\* number of image bands \*/

Image\* byteImage; /\* Use for remapping, if needed \*/

/\*

\*\* Make sure input image is byte data type

\*/

if(getDataType\_Image(inputImage) != CVIP\_BYTE)

{

byteImage = remap\_Image(inputImage,CVIP\_BYTE,0,255);

delete\_Image(inputImage); /\* To avoid memory leaks, delete unused image structure \*/

inputImage = byteImage; /\* Assign remapped image back to original pointer \*/

}

/\*

\*\* Gets the number of image bands (planes)

\*/

no\_of\_bands = getNoOfBands\_Image(inputImage);

/\*

\*\* Gets the number of rows in the input image

\*/

no\_of\_rows = getNoOfRows\_Image(inputImage);

/\*

\*\* Gets the number of columns in the input image

\*/

no\_of\_cols = getNoOfCols\_Image(inputImage);

/\*

\*\* Compares the pixel value at the location (r,c)

\*\* with the threshold value. If it is greater than

\*\* the threshold value it writes 255 at the location

\*\* else it writes 0. Note thta this assumes the input

\*\* image is of data type BYTE.

\*/

for(bands=0; bands < no\_of\_bands; bands++) {

/\*

\*\* reference each band of image data in 2-d matrix form;

\*\* which is used for reading and writing the pixel values

\*/

image = getData\_Image(inputImage, bands);

for(r=0; r < no\_of\_rows; r++) {

for(c=0; c < no\_of\_cols; c++) {

if(image[r][c] > (byte) threshval)

image[r][c] = WHITE\_LAB;

else

image[r][c] = BLACK\_LAB;

}

}

}

return inputImage;

}

Image \*threshold\_getThreadValue(Image \*inputImage){

byte \*\*image; /\* 2-d matrix data pointer \*/

int \*temp;

int t;

unsigned int r, /\* row index \*/

c, /\* column index \*/

bands; /\* band index \*/

unsigned int no\_of\_rows, /\* number of rows in image \*/

no\_of\_cols, /\* number of columns in image \*/

no\_of\_bands; /\* number of image bands \*/

Image\* byteImage; /\* Use for remapping, if needed \*/

/\*

\*\* Make sure input image is byte data type

\*/

if(getDataType\_Image(inputImage) != CVIP\_BYTE)

{

byteImage = remap\_Image(inputImage,CVIP\_BYTE,0,255);

delete\_Image(inputImage); /\* To avoid memory leaks, delete unused image structure \*/

inputImage = byteImage; /\* Assign remapped image back to original pointer \*/

}

/\*

\*\* Gets the number of image bands (planes)

\*/

no\_of\_bands = getNoOfBands\_Image(inputImage);

/\*

\*\* Gets the number of rows in the input image

\*/

no\_of\_rows = getNoOfRows\_Image(inputImage);

/\*

\*\* Gets the number of columns in the input image

\*/

no\_of\_cols = getNoOfCols\_Image(inputImage);

temp = (int\*) malloc((no\_of\_rows \* no\_of\_cols + 1) \* sizeof(int));

temp[0] = (int) (no\_of\_rows \* no\_of\_cols);

for(bands=0; bands < no\_of\_bands; bands++) {

image = getData\_Image(inputImage, bands);

for(r=0; r < no\_of\_rows; r++) {

for(c=0; c < no\_of\_cols; c++) {

temp[r \* no\_of\_cols + c + 1] = (int)image[r][c];

}

}

}

t = getThreadValue(temp);

printf("\tThredhold value is: %d\n",t);

free(temp);

for(bands=0; bands < no\_of\_bands; bands++) {

image = getData\_Image(inputImage, bands);

for(r=0; r < no\_of\_rows; r++) {

for(c=0; c < no\_of\_cols; c++) {

if(image[r][c] > (byte) t)

image[r][c] = WHITE\_LAB;

else

image[r][c] = BLACK\_LAB;

}

}

}

return inputImage;

}

Image\* thredhold\_Mark8(Image\* inputImage)

{

extern int flag;

extern int count;

int counter = 0;

int number = 1;

byte \*\*image;

unsigned int row, col, bands;

unsigned int no\_of\_rows, no\_of\_cols, no\_of\_bands;

Image\* byteImage;

if(getDataType\_Image(inputImage) != CVIP\_BYTE)

{

byteImage = remap\_Image(inputImage,CVIP\_BYTE,0,255);

delete\_Image(inputImage); /\* To avoid memory leaks, delete unused image structure \*/

inputImage = byteImage; /\* Assign remapped image back to original pointer \*/

}

/\*

\*\* Gets the number of image bands (planes)

\*/

no\_of\_bands = getNoOfBands\_Image(inputImage);

/\*

\*\* Gets the number of rows in the input image

\*/

no\_of\_rows = getNoOfRows\_Image(inputImage);

/\*

\*\* Gets the number of columns in the input image

\*/

no\_of\_cols = getNoOfCols\_Image(inputImage);

for(bands=0; bands < no\_of\_bands; bands++) {

byte value = 51;

flag = 0;

counter = 0;

printf("\tThe area of pixel is following:\n");

image = getData\_Image(inputImage, bands);

for(row=0; row < no\_of\_rows; row++) {

for(col=0; col < no\_of\_cols; col++) {

newMark8(image,value,row,col,no\_of\_rows,no\_of\_cols);

if(flag == 1)

{

printf("\t%d : %d\n",number,count);

counter ++;

number ++;

flag = 0;

value ++;

count = 0;

}

}

}

printf("\tTotal is %d objects\n",counter);

}

return inputImage;

}

Image\* thredhold\_Chain8(Image\* inputImage)

{

extern int flag;

extern int count;

byte \*\*image;

unsigned int row, col, bands;

unsigned int no\_of\_rows, no\_of\_cols, no\_of\_bands;

Image\* byteImage;

extern int flag;

flag = 0;

if(getDataType\_Image(inputImage) != CVIP\_BYTE)

{

byteImage = remap\_Image(inputImage,CVIP\_BYTE,0,255);

delete\_Image(inputImage); /\* To avoid memory leaks, delete unused image structure \*/

inputImage = byteImage; /\* Assign remapped image back to original pointer \*/

}

/\*

\*\* Gets the number of image bands (planes)

\*/

no\_of\_bands = getNoOfBands\_Image(inputImage);

/\*

\*\* Gets the number of rows in the input image

\*/

no\_of\_rows = getNoOfRows\_Image(inputImage);

/\*

\*\* Gets the number of columns in the input image

\*/

no\_of\_cols = getNoOfCols\_Image(inputImage);

count = 0;

printf("\tThe chain sequence is following:\n");

for(bands=0; bands < no\_of\_bands; bands++) {

byte value = 50;

byte markValue = 49;

/\*

\*\* reference each band of image data in 2-d matrix form;

\*\* which is used for reading and writing the pixel values

\*/

image = getData\_Image(inputImage, bands);

for(row=0; row < no\_of\_rows; row++) {

for(col=0; col < no\_of\_cols; col++) {

//if(flag == 1)

//return inputImage;

chain8(image,value,markValue,row,col,no\_of\_rows,no\_of\_cols);

}

}

}

return inputImage;

}

Image\* thredhold\_MarkBound(Image\* inputImage)

{

byte \*\*image;

unsigned int row, col, bands;

unsigned int no\_of\_rows, no\_of\_cols, no\_of\_bands;

Image\* byteImage;

extern int flag;

flag = 0;

if(getDataType\_Image(inputImage) != CVIP\_BYTE)

{

byteImage = remap\_Image(inputImage,CVIP\_BYTE,0,255);

delete\_Image(inputImage); /\* To avoid memory leaks, delete unused image structure \*/

inputImage = byteImage; /\* Assign remapped image back to original pointer \*/

}

/\*

\*\* Gets the number of image bands (planes)

\*/

no\_of\_bands = getNoOfBands\_Image(inputImage);

/\*

\*\* Gets the number of rows in the input image

\*/

no\_of\_rows = getNoOfRows\_Image(inputImage);

/\*

\*\* Gets the number of columns in the input image

\*/

no\_of\_cols = getNoOfCols\_Image(inputImage);

for(bands=0; bands < no\_of\_bands; bands++) {

byte value = 50;

/\*

\*\* reference each band of image data in 2-d matrix form;

\*\* which is used for reading and writing the pixel values

\*/

image = getData\_Image(inputImage, bands);

for(row=0; row < no\_of\_rows; row++) {

for(col=0; col < no\_of\_cols; col++) {

if(identifyBound(image,row,col,no\_of\_rows,no\_of\_cols) == 1)

image[row][col] = value;

}

}

for(row=0; row < no\_of\_rows; row++) {

for(col=0; col < no\_of\_cols; col++) {

if(image[row][col] != value)

image[row][col] = 255;

}

}

}

return inputImage;

}

Image \*threshold\_allProcess(Image \*inputImage){

byte \*\*image; /\* 2-d matrix data pointer \*/

int \*temp;

extern int flag;

extern int count;

int counter;

int t;

struct stack\* s;

struct list\* l;

struct report\*\* rep;

struct pixel\* tempp;

unsigned int r, /\* row index \*/

c, /\* column index \*/

bands; /\* band index \*/

unsigned int row, col;

unsigned int no\_of\_rows, /\* number of rows in image \*/

no\_of\_cols, /\* number of columns in image \*/

no\_of\_bands; /\* number of image bands \*/

Image\* byteImage; /\* Use for remapping, if needed \*/

/\*

\*\* Make sure input image is byte data type

\*/

if(getDataType\_Image(inputImage) != CVIP\_BYTE)

{

byteImage = remap\_Image(inputImage,CVIP\_BYTE,0,255);

delete\_Image(inputImage); /\* To avoid memory leaks, delete unused image structure \*/

inputImage = byteImage; /\* Assign remapped image back to original pointer \*/

}

/\*

\*\* Gets the number of image bands (planes)

\*/

no\_of\_bands = getNoOfBands\_Image(inputImage);

/\*

\*\* Gets the number of rows in the input image

\*/

no\_of\_rows = getNoOfRows\_Image(inputImage);

/\*

\*\* Gets the number of columns in the input image

\*/

no\_of\_cols = getNoOfCols\_Image(inputImage);

//get the thredhold value

temp = (int\*) malloc((no\_of\_rows \* no\_of\_cols + 1) \* sizeof(int));

temp[0] = (int) (no\_of\_rows \* no\_of\_cols);

for(bands=0; bands < no\_of\_bands; bands++) {

image = getData\_Image(inputImage, bands);

for(r=0; r < no\_of\_rows; r++) {

for(c=0; c < no\_of\_cols; c++) {

temp[r \* no\_of\_cols + c + 1] = (int)image[r][c];

}

}

}

t = getThreadValue(temp);

printf("\n\n\tThredhold value is: %d\n",t);

free(temp);

//get the thredhold image

for(bands=0; bands < no\_of\_bands; bands++) {

image = getData\_Image(inputImage, bands);

for(r=0; r < no\_of\_rows; r++) {

for(c=0; c < no\_of\_cols; c++) {

if(image[r][c] > (byte) t)

image[r][c] = WHITE\_LAB;

else

image[r][c] = BLACK\_LAB;

}

}

}

view\_Image(inputImage,"threshold");

//get the marked image mark from number 51 begin

s = (struct stack\*) malloc(1 \* sizeof(struct stack));

initialStack(s);

for(bands=0; bands < no\_of\_bands; bands++) {

byte value = 51;

flag = 0;

counter = 0;

//printf("\tThe area of pixel is following:\n");

image = getData\_Image(inputImage, bands);

for(row=0; row < no\_of\_rows; row++) {

for(col=0; col < no\_of\_cols; col++) {

newMark8(image,value,row,col,no\_of\_rows,no\_of\_cols);

if(flag == 1)

{

l = (struct list\*) malloc(1 \* sizeof(struct list));

l->i = count;

stackPush(s,l);

//printf("\t%d : %d\n",number,count);

counter ++;

flag = 0;

value ++;

count = 0;

}

}

}

printf("\n\tTotal is %d objects\n",counter);

}

rep = (struct report\*\*) malloc(counter \* sizeof(struct report\*));

for(r = 0;r < counter;r++)

{

rep[counter - 1 - r] = (struct report\*) malloc(sizeof(struct report));

rep[counter - 1 - r]->area = stackPop(s)->i;

//printf("\t%d: %d\n",r,rep[counter - r]->area);

}

view\_Image(inputImage,"Marked");

//get the bound image mark with value 50

for(bands=0; bands < no\_of\_bands; bands++) {

byte value = 50;

image = getData\_Image(inputImage, bands);

for(row=0; row < no\_of\_rows; row++) {

for(col=0; col < no\_of\_cols; col++) {

if(identifyBound(image,row,col,no\_of\_rows,no\_of\_cols) == 1)

if(image[row][col] > 50 && image[row][col] <= 50 + counter)

image[row][col] = value;

}

}

for(row=0; row < no\_of\_rows; row++) {

for(col=0; col < no\_of\_cols; col++) {

if(image[row][col] != value)

image[row][col] = 255;

}

}

}

view\_Image(inputImage,"BoundMarked");

//get the bound chain chan change the mark to value 49

flag = 0;

count = 0;

for(bands=0; bands < no\_of\_bands; bands++) {

byte value = 50;

byte markValue = 49;

image = getData\_Image(inputImage, bands);

for(row=0; row < no\_of\_rows; row++) {

for(col=0; col < no\_of\_cols; col++) {

tempp = chain8(image,value,markValue,row,col,no\_of\_rows,no\_of\_cols);

if(tempp != NULL)

{

rep[count]->circularity = tempp;

//printf("%d: ",count);

rep[count]->perimeter = calPerimeter(rep[count]->circularity);

rep[count]->diameter = calDiameter(rep[count]->circularity);

//printf("perimeters: %f\n",rep[count]->perimeter);

//printf("diameters: %f\n",rep[count]->diameter);

//printDirection(rep[count]->circularity);

//printf("\n");

//printIndex(rep[count]->circularity);

//printf("\n");

count++;

}

}

}

}

//print the report

printf("\n\tNumber\t\tArea\t\tDiameters\t\tPerimeters\n");

for(r=0;r<counter;r++)

{

printf("\n\t%d\t\t%d\t\t%f\t\t%f\n",r+1,rep[r]->area,rep[r]->diameter,rep[r]->perimeter);

}

printf("\n");

printf("\n\tThe chain direction list is following: \n");

for(r = 0;r<counter;r++)

{

printf("\t%d: ",r + 1);

printDirection(rep[r]->circularity);

printf("\n");

}

printf("\n\tThe chain list index is following: \n");

for(r = 0;r<counter;r++)

{

printf("\t%d: ",r + 1);

printIndex(rep[r]->circularity);

printf("\n");

}

printf("\n\n\n");

return inputImage;

}

double calDiameter(struct pixel\* a)

{

double diameters,rec;

struct pixel\* curr;

struct pixel\*\* temp;

int counter,i,j;

diameters = 0;

rec = 0;

counter = 0;

if(a == NULL)

{

printf("\tThe rep you use is NULL!\n");

return 0;

}

curr = a;

while(curr != NULL)

{

counter ++;

curr = curr->next;

}

temp = (struct pixel\*\*) malloc (counter \* sizeof(pixel\*));

curr = a;

counter = 0;

while(curr != NULL)

{

temp[counter] = curr;

curr = curr->next;

counter ++;

}

for(i=0;i<counter;i++)

{

for(j = i + 1;j<counter;j++)

{

rec = (double)sqrt((temp[i]->i - temp[j]->i) \* (temp[i]->i - temp[j]->i) + (temp[i]->j - temp[j]->j) \* (temp[i]->j - temp[j]->j));

if(rec > diameters)

diameters = rec;

}

}

free(temp);

return diameters;

}

double calPerimeter(struct pixel\* a)

{

double perimeters;

struct pixel\* curr;

perimeters = 0;

if(a == NULL)

{

printf("\tThe rep you use is NULL!\n");

return 0;

}

curr = a;

while(curr != NULL)

{

if(curr->nextDirection == 1 || curr->nextDirection == 3 || curr->nextDirection == 5 || curr->nextDirection == 7)

perimeters = perimeters + 1.414;

else

{

if(curr->nextDirection != 9)

perimeters = perimeters + 1;

}

curr = curr->next;

}

return perimeters;

}

void printDirection(struct pixel\* a)

{

struct pixel\* curr;

if(a == NULL)

{

printf("\tThe rep you use is NULL!\n");

return;

}

curr = a;

while(curr != NULL)

{

printf("%d ",curr->nextDirection);

curr = curr->next;

}

}

void printIndex(struct pixel\* a)

{

struct pixel\* curr;

if(a == NULL)

{

printf("\tThe rep you use is NULL!\n");

return;

}

curr = a;

while(curr != NULL)

{

printf("(%d, %d) ",curr->i,curr->j);

curr = curr->next;

}

}

void initialStack(struct stack\* a)

{

a->num = 0;

a->begin = NULL;

}

void stackPush(struct stack\* s,struct list\* a)

{

if(a == NULL)

{

printf("\t\tPlease do not insert null!\n");

return;

}

s->num = s->num + 1;

a->next = s->begin;

s->begin = a;

}

struct list\* stackPop(struct stack\* s)

{

struct list\* temp;

if(s->begin == NULL)

{

printf("\t\tThe stack is empty!\n");

return NULL;

}

temp = s->begin;

s->num = s->num - 1;

if(s->begin != NULL)

s->begin = s->begin->next;

else

return NULL;

return temp;

}

int getThreadValue(int\* temp)

{

int a1, a2, level,vIndex,i,counter,sum;

int \*v;

int \*\*\*tt;

int \*\*ttBefore;

ttBefore = (int\*\*) malloc(1 \* sizeof(int\*));

ttBefore[0] = temp;

level = 0;

a1 = 0;

vIndex = 0;

a2 = getAverage(temp);

do

{

counter = 0;

sum = 0;

a1 = a2;

vIndex = 2 \* power(2,level);

tt = (int\*\*\*) malloc(vIndex/2 \* sizeof(int\*\*));

v = (int\*) malloc(vIndex/2 \* sizeof(int));

for(i=0;i<vIndex/2;i++)

{

v[i] = getAverage(ttBefore[i]);

tt[i] = part(ttBefore[i],v[i]);

}

for(i=0;i<vIndex/2;i++)

{

sum = sum + getAverage(tt[i][0]) + getAverage(tt[i][1]);

counter = counter + 2;

}

a2 = sum / counter;

level ++;

ttBefore = (int\*\*) malloc(vIndex \* sizeof(int\*));

for(i=0;i<vIndex/2;i++)

{

ttBefore[2\*i] = tt[i][0];

ttBefore[2\*i+1] = tt[i][1];

}

}while(a1-a2>1 || a1-a2<-1);

//printf("\t\tLevel is: %d\n",level);

return a1;

}

int power(int a, int b)

{

int i,c;

c = 1;

if(b == 0)

return 1;

for(i = 0;i<b;i++)

c = c \* a;

return c;

}

int getNextLevelAve(int\* temp)

{

int average,a1,a2;

int \*\*tt;

average = getAverage(temp);

tt = part(temp,average);

a1 = getAverage(tt[0]);

a2 = getAverage(tt[1]);

free(tt[0]);

free(tt[1]);

free(tt);

return (a1 + a2)/2;

}

int getAverage(int\* temp)

{

int t, i;

t = 0;

if(temp[0] == 0)

return 0;

for(i=1;i<=temp[0];i++)

t = t + temp[i];

t = t / temp[0];

return t;

}

int\*\* part(int\* temp,int average)

{

int counter1, counter2,i,t1,t2;

int \*\*t;

t = (int\*\*) malloc(2\*sizeof(int\*));

counter1 = 0;

counter2 = 0;

if(temp[0] == 0)

{

t[0] = (int\*) malloc((counter1 + 1)\*sizeof(int));

t[1] = (int\*) malloc((counter2 + 1)\*sizeof(int));

t[0][0] = 0;

t[1][0] = 0;

return t;

}

for(i=1;i<=temp[0];i++)

{

if(temp[i] >= average)

counter1 ++;

else

counter2 ++;

}

t[0] = (int\*) malloc((counter1 + 1)\*sizeof(int));

t[1] = (int\*) malloc((counter2 + 1)\*sizeof(int));

t1 = 1;

t2 = 1;

t[0][0] = counter1;

t[1][0] = counter2;

for(i=1;i<=temp[0];i++)

{

if(temp[i] >= average)

{

t[0][t1] = temp[i];

t1 ++;

}

else

{

t[1][t2] = temp[i];

t2 ++;

}

}

return t;

}

void mark8( byte\*\* inputImage,

byte value,

unsigned int iseed,

unsigned int jseed,

unsigned int range\_i,

unsigned int range\_j)

{

//iseed and jseed are the position of the first point of an object

//object value is 0 and background value is 0

extern int flag;

extern int count;

if(inputImage[iseed][jseed] != 0)

return;

inputImage[iseed][jseed] = value;

flag = 1;

count ++;

if(iseed > 0)

mark8(inputImage,value,iseed - 1,jseed,range\_i,range\_j);

if(jseed > 0)

mark8(inputImage,value,iseed,jseed - 1,range\_i,range\_j);

if(iseed + 1 < range\_i)

mark8(inputImage,value,iseed + 1,jseed,range\_i,range\_j);

if(jseed + 1 < range\_j)

mark8(inputImage,value,iseed,jseed + 1,range\_i,range\_j);

if(iseed > 0 && jseed > 0)

mark8(inputImage,value,iseed - 1,jseed - 1,range\_i,range\_j);

if(iseed + 1 < range\_i && jseed + 1 < range\_j)

mark8(inputImage,value,iseed + 1,jseed + 1,range\_i,range\_j);

if(iseed > 0 && jseed + 1 < range\_j)

mark8(inputImage,value,iseed - 1,jseed + 1,range\_i,range\_j);

if(iseed + 1 < range\_i && jseed > 0)

mark8(inputImage,value,iseed + 1,jseed - 1,range\_i,range\_j);

}

void newMark8(byte\*\* inputImage,

byte value,

int iseed,

int jseed,

int range\_i,

int range\_j)

{

int i,j,n,m,again;

extern int flag;

extern int count;

if(inputImage[iseed][jseed] != 0)

return;

inputImage[iseed][jseed] = value;

flag = 1;

count ++;

do

{

again = 0;

for(i=0;i<range\_i;i++)

for(j=0;j<range\_j;j++)

if(inputImage[i][j] == value)

for(n=i-1;n<=i+1;n++)

for(m=j-1;m<=j+1;m++)

{

if(range(n,m,range\_i,range\_j) == 0)

continue;

if(inputImage[n][m] == 0)

{

inputImage[n][m] = value;

count++;

again = 1;

}

}

for(i=range\_i-1;i>=0;i--)

for(j=range\_j-1;j>=0;j--)

if(inputImage[i][j] == value)

for(n=i-1;n<=i+1;n++)

for(m=j-1;m<=j+1;m++)

{

if(range(n,m,range\_i,range\_j) == 0)

continue;

if(inputImage[n][m] == 0)

{

inputImage[n][m] = value;

count++;

again = 1;

}

}

}while(again);

}

struct pixel\* chain8(byte\*\* inputImage,

byte value,

byte markValue,

unsigned int iseed,

unsigned int jseed,

unsigned int range\_i,

unsigned int range\_j)

{

struct pixel\* p;

struct pixel\* p1;

struct pixel\* p2;

int di[8],dj[8],i,j,initialDirection,tempi,tempj;

di[0] =0;di[1] = -1;di[2] = -1;di[3] = -1;di[4] = 0;di[5] = 1;di[6] = 1;di[7] = 1;

dj[0] = -1;dj[1] = -1;dj[2] = 0;dj[3] = 1;dj[4] = 1;dj[5] = 1;dj[6] = 0;dj[7] = -1;

tempi = iseed;

tempj = jseed;

i = iseed;

j = jseed;

initialDirection = 1;

if(inputImage[iseed][jseed] != value)

return NULL;

inputImage[iseed][jseed] = markValue;

p = (struct pixel\*) malloc(1 \* sizeof(struct pixel));

p->i = iseed;

p->j = jseed;

p->nextDirection = 9;

p->next = NULL;

//printf("Begin point(%d,%d): ",i,j);

p2 = p;

do

{

int ii;

int boolen;

boolen = 0;

for(ii = 0;ii < 8;ii++)

{

i = tempi + di[(ii + initialDirection) % 8];

j = tempj + dj[(ii + initialDirection) % 8];

if(range((int)i,(int)j,(int)range\_i,(int)range\_j) == 1)

{

if(inputImage[i][j] == value || inputImage[i][j] == markValue)

{

p1 = (struct pixel\*) malloc(1 \* sizeof(struct pixel));

p1->i = i;

p1->j = j;

p1->nextDirection = 9;

p2->nextDirection = (ii + initialDirection)%8;

p1->next = NULL;

p2->next = p1;

p2 = p1;

inputImage[i][j] = markValue;

boolen = 1;

//printf("%d",(ii + initialDirection)%8);

//printf("(%d,%d), ",i,j);

tempi = i;

tempj = j;

break;

}

}

}

if(boolen != 1)

{

//printf(" The chain is break!\n");

return p;

}

initialDirection = (ii + 5 + initialDirection) % 8;

}

while((i != iseed) || (j != jseed));

return p;

//printf("\n");

}

int identifyBound(byte\*\* inputImage,

unsigned int iseed,

unsigned int jseed,

unsigned int range\_i,

unsigned int range\_j)

{

if(inputImage[iseed][jseed] == 255)

return 0;

if(iseed > 0)

{

if(inputImage[iseed - 1][jseed] == 255)

{

return 1;

}

}

else

{

return 1;

}

if(jseed > 0)

{

if(inputImage[iseed][jseed - 1] == 255)

{

return 1;

}

}

else

{

return 1;

}

if(iseed + 1 < range\_i)

{

if(inputImage[iseed + 1][jseed] == 255)

{

return 1;

}

}

else

{

return 1;

}

if(jseed + 1 < range\_j)

{

if(inputImage[iseed][jseed + 1] == 255)

{

return 1;

}

}

else

{

return 1;

}

return 0;

}

int range( int row,

int col,

int range\_i,

int range\_j)

{

if(row < 0 || row > range\_i)

return 0;

if(col < 0 || col > range\_j)

return 0;

return 1;

}