Lab 3: Neighborhoods and Connected Components

Course Title: Image Processing I (Spring 2022)

Course Number: ECE 63700

Instructor: Prof. Charles A. Bouman

Author: **Zhankun Luo**

1. Area Fill

1.1. the gray scale image `img22gd2.tif`

solution

Input Gray Scale Image **img22gd2.tif**

1.2. the image showing the connected set for $s = (67, 45), T = 2$

solution

Connected Set for **s=(67, 45), T=2**

1.3. the image showing the connected set for $s = (67, 45), T = 1$

solution

Connected Set for **s=(67, 45), T=1**

1.4. the image showing the connected set for $s = (67, 45), T = 3$

solution

Connected Set for **s=(67, 45), T=3**

1.5. listing of C code

solution

Functions used in **connect.c**: `ConnectedNeighbors()`, `ConnectedSet()`

Implement them with First-In-First-Out Queue and predefined structs: `Node`, `Queue`

```
struct Node_t {
    struct pixel s;
    struct Node_t* next;
};
typedef struct Node_t Node;

struct Queue_t;
static void enqueue_t(struct Queue_t* p_queue, struct pixel s);
static struct pixel dequeue_t(struct Queue_t* p_queue);
struct Queue_t {
    Node* first;
    Node* last;
    int32_t n;
    void (*enqueue)(struct Queue_t*, struct pixel);
    struct pixel (*dequeue)(struct Queue_t*);
} Queue_default = {NULL, NULL, 0, enqueue_t, dequeue_t};
typedef struct Queue_t Queue;

static void enqueue_t(Queue* p_queue, struct pixel s) {
    Node* last_old = p_queue->last;
    p_queue->last = (Node *)malloc(sizeof(Node));
    p_queue->last->s = s;
    p_queue->last->next = NULL;
    if (p_queue->n == 0) {
        p_queue->first = p_queue->last;
    } else {
        last_old->next = p_queue->last;
    }
    (p_queue->n)++;
}

static struct pixel dequeue_t(Queue* p_queue) {
    if (p_queue->n == 0) { exit(1); }
```

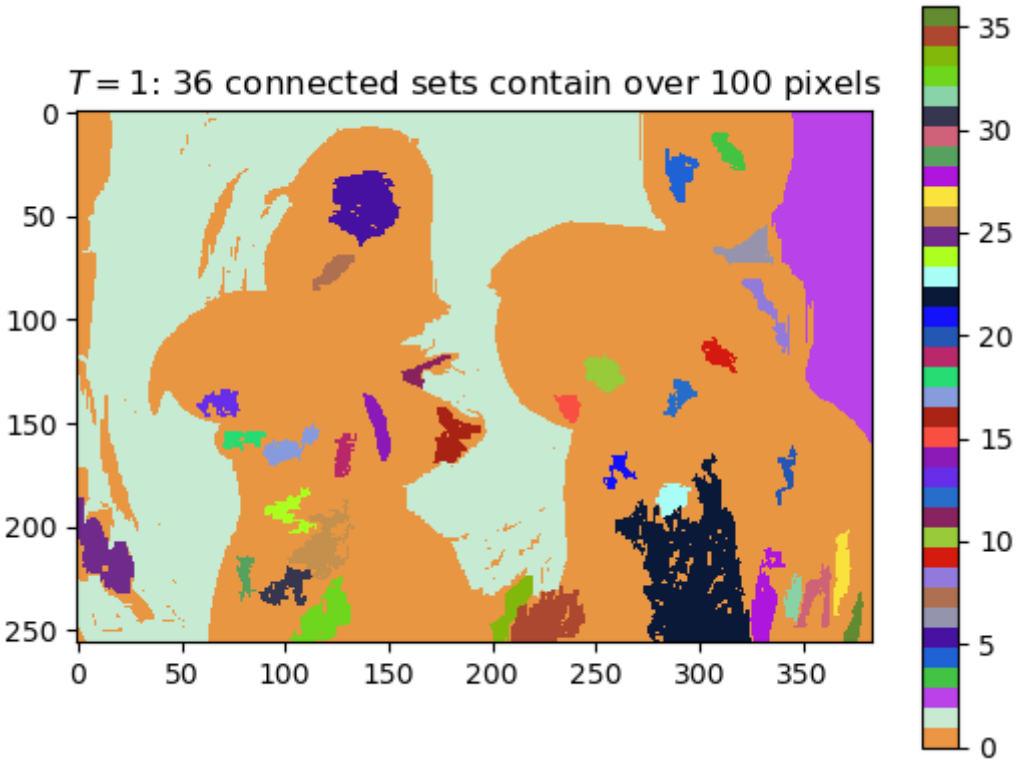
```
    struct pixel s = p_queue->first->s;
    Node* first_tmp = p_queue->first->next;
    free(p_queue->first);
    p_queue->first = first_tmp;
    (p_queue->n)--;
    if (p_queue->n == 0) { p_queue->last == NULL; }
    return s;
}

void ConnectedNeighbors(
    struct pixel s, double T,
    unsigned char **img,
    int width, int height,
    int *M, struct pixel c[4]) {
    (*M) = 0; int t = (int)(T+1e-6);
    int16_t offset[2] = {-1, 1};
    int m, n, m0=s.m, n0=s.n;
    int value = img[m0][n0];
    for (int16_t i=0; i<2; i++) {
        m = m0 + offset[i];
        if (m < 0 || m >= height) continue;
        if (   (((int)(img[m][n0]) - value) <= t)
            && (((int)(img[m][n0]) - value) >= -t)) {
            c[(*M)] = (struct pixel){.m = m, .n = n0};
            (*M)++;
        }
    }
    for (int16_t j=0; j<2; j++) {
        n = n0 + offset[j];
        if (n < 0 || n >= width)  continue;
        if (   (((int)(img[m0][n]) - value) <= t)
            && (((int)(img[m0][n]) - value) >= -t)) {
            c[(*M)] = (struct pixel){.m = m0, .n = n};
            (*M)++;
        }
    }
}

void ConnectedSet(
    struct pixel s, double T,
    unsigned char **img,
    int width, int height,
    int ClassLabel, unsigned int **seg,
```

```
    int *NumConPixels) {
    unsigned char **visit =
        (unsigned char **)get_img(width, height,
                                  sizeof(unsigned char));
    for (int16_t i=0; i < height; i++) {
        memset(visit[i], 0, width*sizeof(unsigned char));
    }
    for (int16_t i = 0; i < height; i++ ) {
        for (int16_t j = 0; j < width; j++ ) {
            if (seg[i][j] != 0) visit[i][j] = 1;
        }
    }
    Queue q_boundary = Queue_default;
    Queue *this = &q_boundary;
    q_boundary.enqueue(this, s);
    visit[s.m][s.n] = 2;
    int num_connect;
    struct pixel neighbor_connect[4], s_boundary;
    (*NumConPixels) = 0;
    while (q_boundary.n != 0) {
        s_boundary = q_boundary.dequeue(this);
        seg[s_boundary.m][s_boundary.n] = ClassLabel;
        visit[s_boundary.m][s_boundary.n] = 1;
        (*NumConPixels)++;
        ConnectedNeighbors(s_boundary, T, img,
                           width, height,
                           &num_connect, neighbor_connect);
        if (num_connect == 0) continue;
        for (int16_t i=0; i<num_connect; i++) {
            if (visit[neighbor_connect[i].m]
                    [neighbor_connect[i].n] == 0) {
                q_boundary.enqueue(this, neighbor_connect[i]);
                visit[neighbor_connect[i].m]
                    [neighbor_connect[i].n] = 2;
            }
        }
    }
    free_img( (void**)visit );
}
```

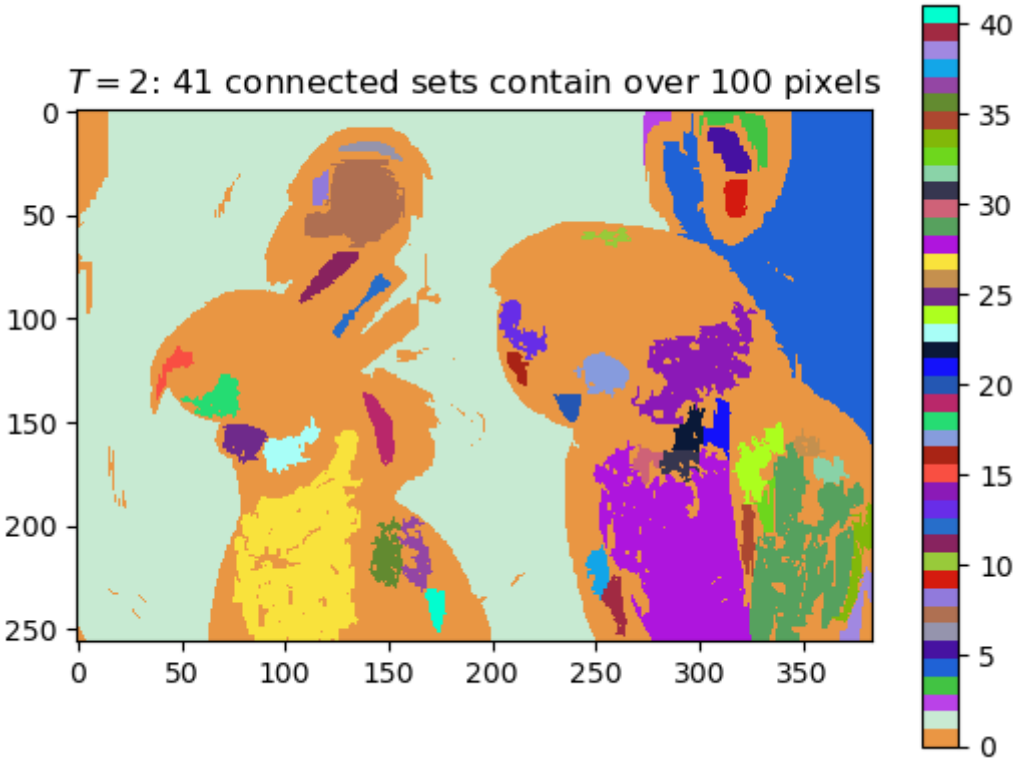
2. Image Segmentation

2.1. the randomly colored segmentation for $T = 1, 2, 3$

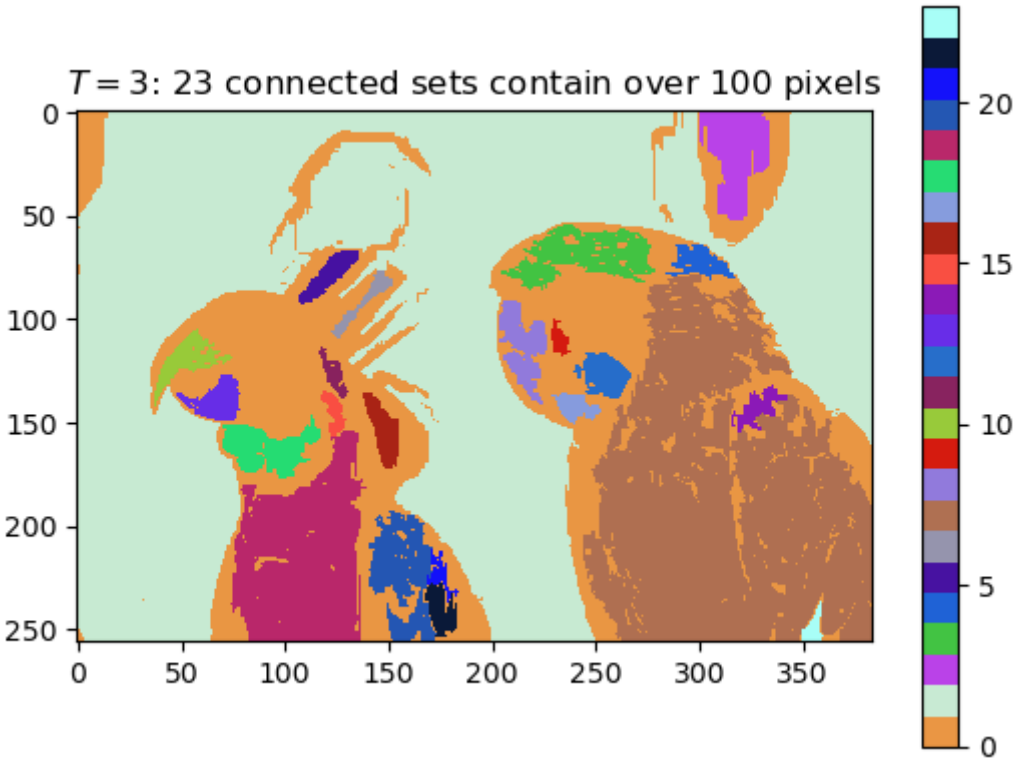
solution



Randomly Colored Segmentation for **T=1**



Randomly Colored Segmentation for **T=2**



Randomly Colored Segmentation for **T=3**

2.2. the number of regions generated for each of $T = 1, 2, 3$

solution

T	**1**	**2**	**3**
Number of Connected Sets > 100 pixels	36	41	23
Number of Total Connected Sets	27654	16747	11192

4.7. listing of C code

solution

Function used in **connect.c**: `Segment()`

Implement it based the predefined function: `ConnectedSet()`

```
void Segment(
    double T, int num_pixel_min,
    unsigned char **img,
    unsigned int **seg,
    int width, int height,
    int* ptr_num_connect_set) {
    for (int16_t i=0; i < height; i++) {
        memset(seg[i], 0, width*sizeof(unsigned int));
    }
    int32_t size_list = 256;
    int* list_num_pixel = (int*) malloc(size_list*sizeof(int));
    int num_connect_set_all = 0;
    struct pixel s;
    for (int16_t i=0; i<height; i++) {
        for (int16_t j=0; j<width; j++) {
            if (seg[i][j] == 0) {
                if (num_connect_set_all >= size_list) {
                    size_list <<= 1;
                    list_num_pixel =
                        (int*) realloc(list_num_pixel,
                                       size_list*sizeof(int));
                }
                s = (struct pixel){.m = i, .n = j};
                num_connect_set_all++;
                ConnectedSet(s, T, img, width, height,
                    num_connect_set_all, seg,
                    list_num_pixel+(num_connect_set_all-1));
            }
        }
    }
    *ptr_num_connect_set = 0;
    printf("num_connect_set_all:%d\n", num_connect_set_all);
    int16_t* list_class_label =
```

```
        (int16_t*) malloc(num_connect_set_all*sizeof(int16_t));
    memset(list_class_label, 0, num_connect_set_all*sizeof(int16_t));
    for (int32_t i=0; i<num_connect_set_all; i++) {
        if (list_num_pixel[i] > num_pixel_min) {
            (*ptr_num_connect_set)++;
            list_class_label[i] = (*ptr_num_connect_set);
        }
    }
    for (int16_t i=0; i<height; i++) {
        for (int16_t j=0; j<width; j++) {
            seg[i][j] = list_class_label[seg[i][j]-1];
        }
    }
    free(list_num_pixel);
    free(list_class_label);
}
```

Appendix

C codes for connected components: `connect.h`, `connect.c`

connect.h

```
#ifndef _CONNECT_H_
#define _CONNECT_H_

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include <math.h>
#include "allocate.h"
#include "typeutil.h"
#include "tiff.h"

struct pixel { int m, n; };
void ConnectedNeighbors(
    struct pixel s,
    double T,
    unsigned char **img,
    int width,
    int height,
    int *M,
    struct pixel c[4]
);
void ConnectedSet(
    struct pixel s,
    double T,
    unsigned char **img,
    int width,
    int height,
    int ClassLabel,
    unsigned int **seg,
    int *NumConPixels
);
void Segment(
    double T, int num_pixel_min,
```

```
    unsigned char **img,
    unsigned int **seg,
    int width, int height,
    int* ptr_num_connect_set
);
void print_out_reverse(unsigned int **array, int16_t H, int16_t W);
void assign_img2arr(struct TIFF_img *img, unsigned char **array);
void assign_arr2img(unsigned int **array, struct TIFF_img *img);
#endif /* _CONNECT_H_ */
```

connect.c

```
#include "../include/connect.h"

struct Node_t {
    struct pixel s;
    struct Node_t* next;
};
typedef struct Node_t Node;

struct Queue_t;
static void enqueue_t(struct Queue_t* p_queue, struct pixel s);
static struct pixel dequeue_t(struct Queue_t* p_queue);
struct Queue_t {
    Node* first;
    Node* last;
    int32_t n;
    void (*enqueue)(struct Queue_t*, struct pixel);
    struct pixel (*dequeue)(struct Queue_t*);
} Queue_default = {NULL, NULL, 0, enqueue_t, dequeue_t};
typedef struct Queue_t Queue;

static void enqueue_t(Queue* p_queue, struct pixel s) {
    Node* last_old = p_queue->last;
    p_queue->last = (Node *)malloc(sizeof(Node));
    p_queue->last->s = s;
    p_queue->last->next = NULL;
    if (p_queue->n == 0) {
        p_queue->first = p_queue->last;
    } else {
        last_old->next = p_queue->last;
    }
```

```
    (p_queue->n)++;
}

static struct pixel dequeue_t(Queue* p_queue) {
    if (p_queue->n == 0) { exit(1); }
    struct pixel s = p_queue->first->s;
    Node* first_tmp = p_queue->first->next;
    free(p_queue->first);
    p_queue->first = first_tmp;
    (p_queue->n)--;
    if (p_queue->n == 0) { p_queue->last == NULL; }
    return s;
}

void ConnectedNeighbors(
    struct pixel s, double T,
    unsigned char **img,
    int width, int height,
    int *M, struct pixel c[4]) {
    (*M) = 0; int t = (int)(T+1e-6);
    int16_t offset[2] = {-1, 1};
    int m, n, m0=s.m, n0=s.n;
    int value = img[m0][n0];
    for (int16_t i=0; i<2; i++) {
        m = m0 + offset[i];
        if (m < 0 || m >= height) continue;
        if (   (((int)(img[m][n0]) - value) <= t)
            && (((int)(img[m][n0]) - value) >= -t)) {
            c[(*M)] = (struct pixel){.m = m, .n = n0};
            (*M)++;
        }
    }
    for (int16_t j=0; j<2; j++) {
        n = n0 + offset[j];
        if (n < 0 || n >= width)  continue;
        if (   (((int)(img[m0][n]) - value) <= t)
            && (((int)(img[m0][n]) - value) >= -t)) {
            c[(*M)] = (struct pixel){.m = m0, .n = n};
            (*M)++;
        }
    }
}
```

```
void ConnectedSet(
    struct pixel s, double T,
    unsigned char **img,
    int width, int height,
    int ClassLabel, unsigned int **seg,
    int *NumConPixels) {
    unsigned char **visit =
        (unsigned char **)get_img(width, height,
                                  sizeof(unsigned char));
    for (int16_t i=0; i < height; i++) {
        memset(visit[i], 0, width*sizeof(unsigned char));
    }
    for (int16_t i = 0; i < height; i++ ) {
        for (int16_t j = 0; j < width; j++ ) {
            if (seg[i][j] != 0) visit[i][j] = 1;
        }
    }
    Queue q_boundary = Queue_default;
    Queue *this = &q_boundary;
    q_boundary.enqueue(this, s);
    visit[s.m][s.n] = 2;
    int num_connect;
    struct pixel neighbor_connect[4], s_boundary;
    (*NumConPixels) = 0;
    while (q_boundary.n != 0) {
        s_boundary = q_boundary.dequeue(this);
        seg[s_boundary.m][s_boundary.n] = ClassLabel;
        visit[s_boundary.m][s_boundary.n] = 1;
        (*NumConPixels)++;
        ConnectedNeighbors(s_boundary, T, img,
                           width, height,
                           &num_connect, neighbor_connect);
        if (num_connect == 0) continue;
        for (int16_t i=0; i<num_connect; i++) {
            if (visit[neighbor_connect[i].m]
                [neighbor_connect[i].n] == 0) {
                q_boundary.enqueue(this, neighbor_connect[i]);
                visit[neighbor_connect[i].m]
                    [neighbor_connect[i].n] = 2;
            }
        }
    }
    free_img( (void**)visit );
```

```
}

void Segment(
    double T, int num_pixel_min,
    unsigned char **img,
    unsigned int **seg,
    int width, int height,
    int* ptr_num_connect_set) {
    for (int16_t i=0; i < height; i++) {
        memset(seg[i], 0, width*sizeof(unsigned int));
    }
    int32_t size_list = 256;
    int* list_num_pixel = (int*) malloc(size_list*sizeof(int));
    int num_connect_set_all = 0;
    struct pixel s;
    for (int16_t i=0; i<height; i++) {
        for (int16_t j=0; j<width; j++) {
            if (seg[i][j] == 0) {
                if (num_connect_set_all >= size_list) {
                    size_list <<= 1;
                    list_num_pixel =
                        (int*) realloc(list_num_pixel,
                                       size_list*sizeof(int));
                }
                s = (struct pixel){.m = i, .n = j};
                num_connect_set_all++;
                ConnectedSet(s, T, img, width, height,
                    num_connect_set_all, seg,
                    list_num_pixel+(num_connect_set_all-1));
            }
        }
    }
    *ptr_num_connect_set = 0;
    printf("num_connect_set_all:%d\n", num_connect_set_all);
    int16_t* list_class_label
        = (int16_t*) malloc(num_connect_set_all*sizeof(int16_t));
    memset(list_class_label, 0, num_connect_set_all*sizeof(int16_t));
    for (int32_t i=0; i<num_connect_set_all; i++) {
        if (list_num_pixel[i] > num_pixel_min) {
            (*ptr_num_connect_set)++;
            list_class_label[i] = (*ptr_num_connect_set);
        }
    }
```

```
    for (int16_t i=0; i<height; i++) {
        for (int16_t j=0; j<width; j++) {
            seg[i][j] = list_class_label[seg[i][j]-1];
        }
    }
    free(list_num_pixel);
    free(list_class_label);
}

void print_out_reverse(unsigned int **array, int16_t H, int16_t W) {
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            array[i][j] = ((array[i][j] == 0)? 255: 0);
        }
    }
}

void assign_img2arr(struct TIFF_img *img, unsigned char **array) {
    int16_t W, H;
    W = img->width; H = img->height;
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            array[i][j] = img->mono[i][j];
        }
    }
}

void assign_arr2img(unsigned int **array, struct TIFF_img *img) {
    int16_t W, H;
    W = img->width; H = img->height;
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            int16_t t = array[i][j];
            t = (t < 0)? 0: (t > 255)? 255: t;
            img->mono[i][j] = t;
        }
    }
}
```

C codes for solutions

solution to section 1: `soln_1.c`

```c
/* ECE 637 Image Processing I, Spring 2022
 * @author: Zhankun Luo, luo333@purdue.edu
 * lab 3: Neighborhoods and Connected Components
 * solution to section 1
 * run it with: ./soln_1 img22gd2.tif
 * or run linkedlist version: ./soln_1_linklist img22gd2.tif
 **/

#include "../include/tiff.h"
#include "../include/allocate.h"
#include "../include/typeutil.h"
#include "../include/connect.h"
void error(char *name) {
    printf("usage:  %s  image.tif \n\n",name);
    exit(1);
}

void fill_area(struct TIFF_img img, struct TIFF_img img_out,
                struct pixel s, double T, char index) {
    /* copy image to array */
    int W = img.width, H = img.height;
    unsigned char **arr =
        (unsigned char **)get_img(W, H, sizeof(unsigned char));
    unsigned int **arr_out =
        (unsigned int **)get_img(W, H, sizeof(unsigned int));
    for (int16_t i=0; i < H; i++) {
        memset(arr_out[i], 0, W*sizeof(unsigned int));
    }
    int ClassLabel = 255, NumConPixels;
    assign_img2arr(&img, arr);
    /* fill the area of connected neighbors */
    ConnectedSet(s, T, arr, W, H,
                 ClassLabel, arr_out, &NumConPixels);
    print_out_reverse(arr_out, H, W);
    /* clip to [0, 255] then assign values of arrays to image */
    assign_arr2img(arr_out, &img_out);
```

```
    /* open output image file */
    FILE *fp;
    char path_out[50];
    sprintf(path_out, "../result/fig_1_%c.tif", index);
    if ( ( fp = fopen ( path_out, "wb" ) ) == NULL ) {
        fprintf( stderr, "cannot open TIFF file\n");
        exit( 1 );
    }
    /* write output image */
    if ( write_TIFF( fp, &img_out ) ) {
        fprintf( stderr, "error writing TIFF file\n");
        exit( 1 );
    }
    /* close output image file */
    fclose( fp );
    free_img( (void**)arr );
    free_img( (void**)arr_out );
}

int main(int argc, char **argv) {
    if ( argc != 2 ) error( argv[0] );
    FILE *fp;
    struct TIFF_img img, img_out;
    unsigned int **arr, **arr_out;
    int16_t W, H;
    /* open image file */
    if ( ( fp = fopen( argv[1], "rb" ) ) == NULL ) {
        fprintf( stderr, "cannot open file %s\n", argv[1] );
        exit( 1 );
    }
    /* read image */
    if ( read_TIFF( fp, &img ) ) {
        fprintf( stderr, "error reading file %s\n", argv[1] );
        exit( 1 );
    }
    /* close image file */
    fclose( fp );
    /* check the type of image data: grayscale */
    if ( img.TIFF_type != 'g' ) {
        fprintf( stderr, "error:  image must be grayscale image\n" );
        exit( 1 );
    }
    W = img.width; H = img.height;
```

```
    get_TIFF( &img_out, H, W, 'g' );
    /* search the connected set for T = 2, 1, 3 */
    struct pixel s = {.m=67, .n=45};
    double list_T[3] = {2, 1, 3};
    char list_index[3] = {'2', '3', '4'};
    for (int16_t i=0; i < 3; i++) {
        fill_area(img, img_out, s, list_T[i], list_index[i]);
    }
    /* de-allocate memory */
    free_TIFF( &(img) );
    free_TIFF( &(img_out) );
    return(0);
}
```

solution to section 2: `soln_2.c`

```c
/* ECE 637 Image Processing I, Spring 2022
 * @author: Zhankun Luo, luo333@purdue.edu
 * lab 3: Neighborhoods and Connected Components
 * solution to section 2
 * run it with: ./soln_2 img22gd2.tif
 * or run linkedlist version: ./soln_2_linklist img22gd2.tif
 **/

#include "../include/tiff.h"
#include "../include/allocate.h"
#include "../include/typeutil.h"
#include "../include/connect.h"
void error(char *name) {
    printf("usage:  %s  image.tif \n\n",name);
    exit(1);
}

void segment_image(struct TIFF_img img, struct TIFF_img img_out,
                double T, int num_pixel_min, char index) {
    /* copy image to array */
    int W = img.width, H = img.height;
    unsigned char **arr =
        (unsigned char **)get_img(W, H, sizeof(unsigned char));
    unsigned int **arr_out =
        (unsigned int **)get_img(W, H, sizeof(unsigned int));
    assign_img2arr(&img, arr);
    /* segmentation of connected sets > certain pixels */
    int num_connect_set;
    Segment(T, num_pixel_min, arr, arr_out,
            W, H, &num_connect_set);
    printf("num_connect_set:%d\n", num_connect_set);
    /* clip to [0, 255] then assign values of arrays to image */
    assign_arr2img(arr_out, &img_out);
    /* open output image file */
    FILE *fp;
    char path_out[50];
    sprintf(path_out, "../result/fig_2_1%c.tif", index);
    if ( ( fp = fopen ( path_out, "wb" ) ) == NULL ) {
        fprintf( stderr, "cannot open TIFF file\n");
```

```
        exit( 1 );
    }
    /* write output image */
    if ( write_TIFF( fp, &img_out ) ) {
        fprintf( stderr, "error writing TIFF file\n");
        exit( 1 );
    }
    /* close output image file */
    fclose( fp );
    free_img( (void**)arr );
    free_img( (void**)arr_out );
}

int main(int argc, char **argv) {
    if ( argc != 2 ) error( argv[0] );
    FILE *fp;
    struct TIFF_img img, img_out;
    unsigned int **arr, **arr_out;
    int16_t W, H;
    /* open image file */
    if ( ( fp = fopen( argv[1], "rb" ) ) == NULL ) {
        fprintf( stderr, "cannot open file %s\n", argv[1] );
        exit( 1 );
    }
    /* read image */
    if ( read_TIFF( fp, &img ) ) {
        fprintf( stderr, "error reading file %s\n", argv[1] );
        exit( 1 );
    }
    /* close image file */
    fclose( fp );
    /* check the type of image data: grayscale */
    if ( img.TIFF_type != 'g' ) {
        fprintf( stderr, "error:  image must be grayscale image\n" );
        exit( 1 );
    }
    W = img.width; H = img.height;
    get_TIFF( &img_out, H, W, 'g' );
    /* image segmentation for T = 2, 1, 3 */
    double list_T[3] = {1, 2, 3};
    char list_index[3] = {'a', 'b', 'c'};
    int num_pixel_min = 100;
    for (int16_t i=0; i < 3; i++) {
```

```
        segment_image(img, img_out,
                      list_T[i], num_pixel_min, list_index[i]);
    }
    /* de-allocate memory */
    free_TIFF( &(img) );
    free_TIFF( &(img_out) );
    return(0);
}
```

Python codes for visualizations

visualization to section 2: `vis_2.py`

```
import sys
from os.path import dirname
sys.path.insert(0, dirname(dirname(__file__)))
import numpy as np
from numpy import concatenate, ones
from numpy.random import rand, seed
from PIL import Image
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

def set_colormap(x: np.ndarray, T: int, state: int = 314) -> None:
    N = np.max(x)+1 # number of region: connected set + background
    seed(state)
    cmap = concatenate((rand(N, 3), ones((N, 1))), axis=-1)
    cmap = ListedColormap(cmap)
    plt.imshow(x, cmap=cmap, interpolation='none')
    plt.colorbar()
    plt.title(r"$T=${}: {} connected sets contain over 100 pixels"
              .format(T, N-1))

if __name__ == "__main__":
    for T, index in list(zip([1, 2, 3], ['a', 'b', 'c'])):
        path_in  = "result/fig_2_1" + index + ".tif"
        path_out = "result/fig_2_1" + index + ".png"
        x = np.array(Image.open(path_in))
        set_colormap(x, T)
        plt.savefig(path_out,  bbox_inches='tight')
        plt.show()
```