ECE637: Laboratory 3, Neighborhoods and Connected Components

Joseph K. Kulikowski
Purdue University

February 1, 2025

Contents

List of Figures

Listings

List of Tables

1 Section 1 Report: Area Fill

1.1 A print out the image img22gd2.tif

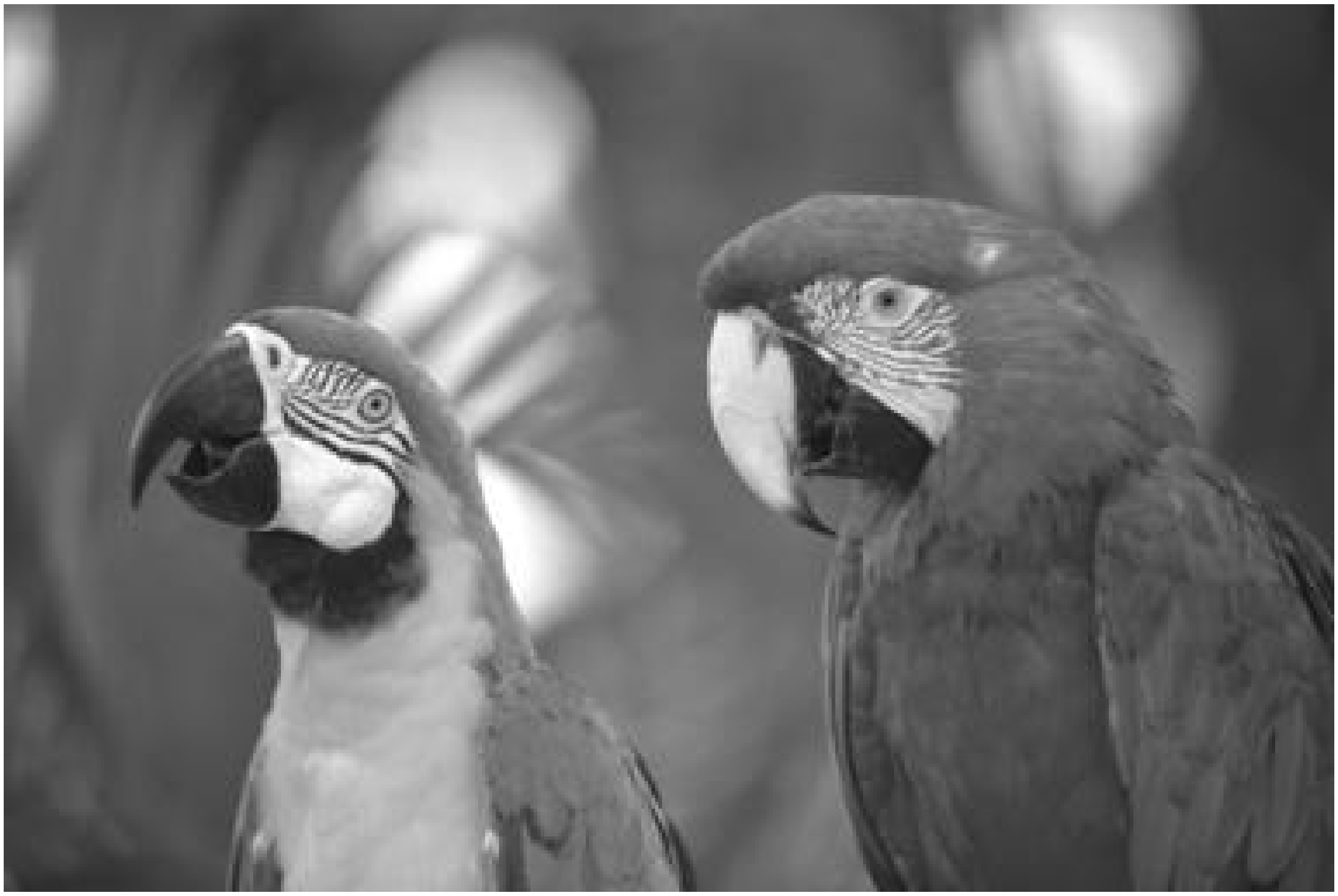

Figure 1: The grayscale image *img22gd2.tif*.

1.2 A print out of the image showing the connected set for s = (67, 45), and T = 2.

It is important to note that s = (67,45) corresponds to pixel row 45, column 67 for the image. As you can see, the connected set to that pixel is colored black, as instructed. It should be in the top left region.

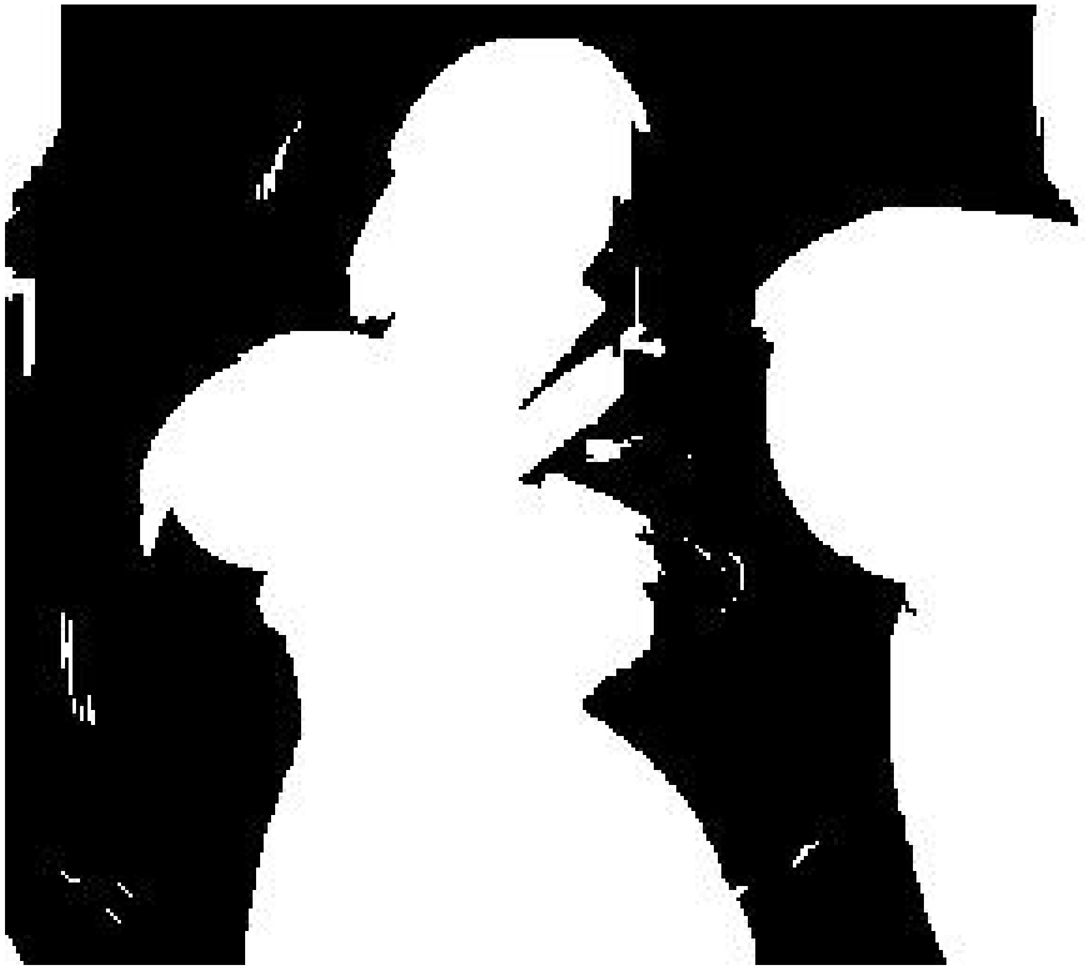

Figure 2: A print out of the image showing the connected set for s = (67, 45), and T = 2.

1.3 A print out of the image showing the connected set for s = (67, 45), and T = 1.

It is important to note that s = (67,45) corresponds to pixel row 45, column 67 for the image. As you can see, there are only 3 pixels in the connected set. As a result of the tightening of the threshold, we didn't escape a local minima present at this pixel. You can ignore the noise that is present around the black rectangle due to jpeg conversion.

Figure 3: A print out of the image showing the connected set for s = (67, 45), and T = 1.

1.4 A print out of the image showing the connected set for s = (67, 45), and T = 3.

It is important to note that s = (67,45) corresponds to pixel row 45, column 67 for the image. As you can see, the loosening of the threshold allowed the background to sneak closer to both birds.

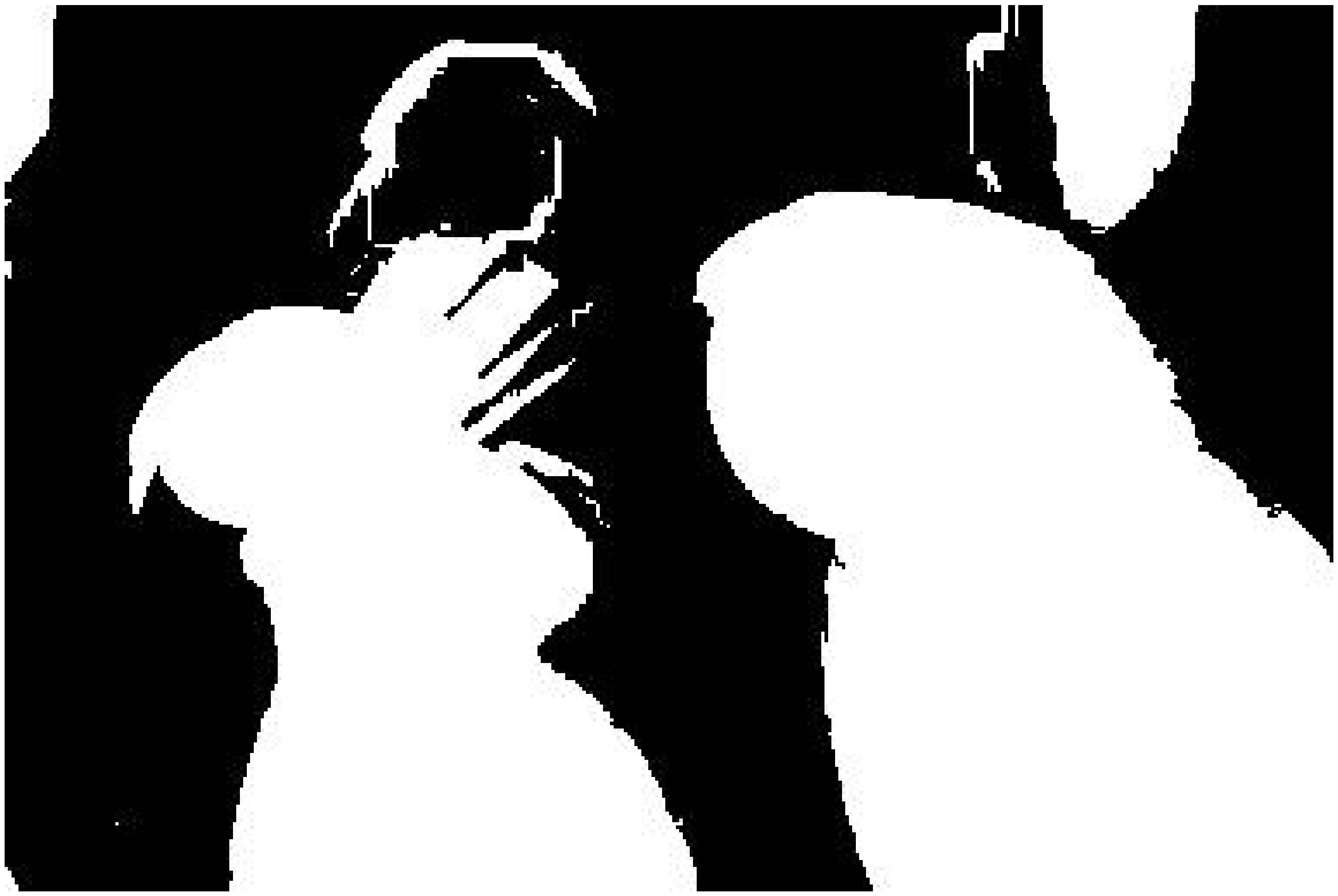

Figure 4: A print out of the image showing the connected set for s = (67, 45), and T = 3.

1.5 A listing of my C code.

To perform the above search for the connected set, the main function, sec_1.c1 is shown below. It does things like accept variable arguments in for the starting pixel and threshold. It leverages a couple of functions: ConnectedSet, which finds the set connected to the pixel, and ConnectedNeighbors, which finds the neighbors connected to a pixel. These are shown in connected_neighbors_sets.c2. In connected_neighbors_sets.c2, you will see the same functions with extensions "2D". These are *untested* attempts at a generalization of the routines to any neighborhood size.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "allocate.h"
#include "tiff.h"
#include "connected_neighbors_sets.h"

void error(char *name);

int main(int argc, char **argv){
    FILE *fp;
    struct TIFF_img input_img, seg_img;
    struct pixel seed;
    int i, j, T, ClassLabel, NumConPixels=0;
    unsigned int **seg;
    char outfile[100];

    /*
    5 args. 1st is this file, 2nd is image to segment,
    3rd is row of seed pixel. 4th is col of seed.
    5th is threshold.
    */
    if(argc != 5) error(argv[0]);

    /* 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, &input_img)){
    fprintf(stderr, "error reading file %s\n", argv[1]);
    exit(1);
    }
    if(input_img.TIFF_type != 'g'){
    fprintf(stderr, "error:  image must be grayscale.\n");
    exit(1);
    }
    /* close image file */
    fclose(fp);

    /* Read in seed pixel and threshold. */
    seed.m = atoi(argv[2]);
    seed.n = atoi(argv[3]);
    T = atoi(argv[4]);

    /* Initialize the connected set to 0, as described in the algorithm */
    seg = multialloc(sizeof(unsigned int), 2, input_img.height, input_img.width);
    for(i=0; i<input_img.height; i++){
        for(j=0; j<input_img.width; j++){
            seg[i][j] = 0;
        }
    }
    ClassLabel = 1;
    ConnectedSet(seed, T, input_img.mono, input_img.width, input_img.height,
                 ClassLabel, seg, &NumConPixels);
    printf("Number of connected pixels: %d\n", NumConPixels);
    get_TIFF(&seg_img, input_img.height, input_img.width, 'g');
    /* Flip-flop and send seg to seg_img as described. */
    for(i=0; i<seg_img.height; i++){
        for(j=0; j<seg_img.width; j++){
            if(seg[i][j]==ClassLabel){
                /* n.b 0=black, 255=white. */
                seg_img.mono[i][j] = 0;
            }else{
                seg_img.mono[i][j] = 255;
            }
        }
    }

    /* Make a sensible output filename. */
    snprintf(outfile, sizeof(outfile), "images/s1_%d_%d_%d.tif", seed.m, seed.n, T);

        /* Try and open the file, error if can't. */
    if((fp = fopen(outfile, "wb")) == NULL){
        fprintf(stderr, "cannot open file %s\n", outfile);
        exit(1);
    }

    /* Try and write the file, error if can't. */
    if(write_TIFF(fp, &seg_img)){
        fprintf(stderr, "error writing TIFF file %s\n", outfile);
        exit(1);
    }
    /* Close the file. */
    fclose(fp);

    multifree(seg, 2);
    free_TIFF(&seg_img);
    return 0;
}

void error(char *name){
    printf("usage:  %s  image.tiff row col threshold\n\n",name);
    printf("this program reads in a grayscale TIFF image.\n");
    printf("It finds the connected sets from a pixel at \n");
    printf("row col for a threshold.\n");
    exit(1);
}
```

Listing 1: sec_1.c

```c
#include <stdlib.h>
#include "allocate.h"
struct pixel{
    int m,n;
    /* m rows, n columns */
};

void ConnectedNeighbors(struct pixel s, double T, unsigned char **img,
                        int width, int height, int *M, struct pixel c[4]){
    /*
    Author: Joseph K. Kulikowski
    Determine the connected neighbors, c[n] | n<=4, to a pixel p within an
    image img of width and height, alongside the number of connected
    neighbors n.
    */

    int i, fb_conds, abs_diff; /* iterator, pixel in bounds, absolute difference*/
    struct pixel neighbor;
    int nh_offsets[4][2] = {
        {-1, 0},  /* above pixel. */
        {1, 0}, /* below pixel. */
        {0, -1}, /* left pixel. */
        {0, 1}, /* right pixel. */
    };
    /* Initialize number of connected neighborhoods to zero. */
    *M = 0;
    for(i=0; i<4; i++){
        /* find position of neighbor given offset. */
        neighbor.m = s.m + nh_offsets[i][0];
        neighbor.n = s.n + nh_offsets[i][1];
        /* Determine if the neighbor exists in the image */
        fb_conds = (neighbor.m < height) &&
                   (neighbor.m >=0) &&
                   (neighbor.n < width) &&
                   (neighbor.n >= 0);
        if(fb_conds){
            abs_diff = abs(img[s.m][s.n] - img[neighbor.m][neighbor.n]);
            /* Do we meet the threshold? */
            if(abs_diff <=T){
                /* If so, neighbor is connected. */
                c[*M].m = neighbor.m;
                c[*M].n = neighbor.n;
                /* Increment number of neighbors connected. */
                (*M)++;
            }
        }
    }

}

void ConnectedSet(struct pixel s, double T, unsigned char **img,
                  int width, int height, int ClassLabel,
                  unsigned int **seg, int *NumConPixels){
    /* Author: Joseph K. Kulikowski */
    /* a list of pixels B which are known to be connected to s0, but have not yet been searched */
    /* n.b. ANY pixel in the image COULD BE connected to s0. */
    int i, B_count=0;
    int M; /* connected neighbors */
    struct pixel loop_pixel, neighbor, neighbors[4], *B;
    B = (struct pixel *)multialloc(sizeof(struct pixel), 1, width * height);

    /* Initialize seed pixel to ClassLabel */
    seg[s.m][s.n] = ClassLabel;
    /* Put our seed pixel into the boundary set. (seed is connected to itself.) */
    B[B_count] = s;
    /* We now have one element in B, and 1 connected pixel. */
    B_count += 1;
    (*NumConPixels)++;

    while (B_count > 0){
        /* Decrement until we run out of pixels. */
        B_count -= 1;
        /* We expect to add ConnectedNeighbors to boundary set. */
        loop_pixel = B[B_count];
        M = 0; /* No connected neighbors yet! */
        /* Determine M neighbors, locations in neighbors */
        ConnectedNeighbors(loop_pixel, T, img, width, height, &M, neighbors);
        /* Loop through connected neighbors, add to boundary set */
        /* set seg[i][j] = ClassLabel where connected */
        for(i=0; i<M; i++){
            neighbor = neighbors[i];
            /* If neighbor hasn't been marked, it hasn't been added. */
            if(seg[neighbor.m][neighbor.n]!=ClassLabel){
                /* Mark seg as ClassLabel */
                seg[neighbor.m][neighbor.n] = ClassLabel;
                /* Add neighbor to boundary set! */
                B[B_count] = neighbor;
                /* Add to our B_count. */
                B_count += 1;
                /* increment our NumConPixels */
                (*NumConPixels)++;
            }
        }
    }
    multifree(B, 1);
}

void ConnectedNeighbors2D(struct pixel s, double T, unsigned char **img,
                          int n_neighbors, int **nh_offsets,
                          int width, int height, int *M, struct pixel c[]){
    /*
    Author: Joseph K. Kulikowski
    Determine the connected neighbors, c[n] | n<=n_neighbors, to a pixel p within an
    image img of width and height, alongside the number of connected
    neighbors n, using neighbors shifted by offsets nh_offsets[n_neighbors][2].
    */
    int i, fb_conds, abs_diff; /* iterator, pixel in bounds, absolute difference*/
    struct pixel neighbor;
    *M = 0;
    for(i=0; i<n_neighbors; i++){
        /* find position of neighbor given offset. */
        neighbor.m = s.m + nh_offsets[i][0];
        neighbor.n = s.n + nh_offsets[i][1];
        /* Determine if the neighbor exists in the image */
        fb_conds = (neighbor.m < height) &&
                   (neighbor.m >=0) &&
                   (neighbor.n < width) &&
                  (neighbor.n >= 0);
        if(fb_conds){
            abs_diff = abs(img[s.m][s.n] - img[neighbor.m][neighbor.n]);
            /* Do we meet the threshold? */
            if(abs_diff<=T){
                /* If so, neighbor is connected. */
                c[*M].m = neighbor.m;
                c[*M].n = neighbor.n;
                /* Increment number of neighbors connected. */
                (*M)++;
            }
        }
    }
}

void ConnectedSet2D(struct pixel s, double T, unsigned char **img,
                  int width, int height, int ClassLabel,
                  int n_neighbors, int **nh_offsets,
                  unsigned int **seg, int *NumConPixels){
    /* Author: Joseph K. Kulikowski */
    /* a list of pixels B which are known to be connected to s0, but have not yet been searched */
    /* n.b. ANY pixel in the image COULD BE connected to s0. */
    struct pixel loop_pixel, neighbor, *neighbors, *B;
    int i, B_count=0;
    int M; /* connected neighbors */
    neighbors = (struct pixel *)multialloc(sizeof(struct pixel), 1, n_neighbors);
    B = (struct pixel *)multialloc(sizeof(struct pixel), 1, width * height);

    /* Initialize seed pixel to ClassLabel */
    seg[s.m][s.n] = ClassLabel;
    /* Put our seed pixel into the boundary set. (seed is connected to itself.) */
    B[B_count] = s;
    /* We now have one element in B, and 1 connected pixel. */
    B_count += 1;
    (*NumConPixels)++;

    while (B_count > 0){
        /* Decrement until we run out of pixels. */
        B_count -= 1;
        /* The loop pixel should be removed from boundary set after loop. */
        /* We expect to add ConnectedNeighbors to boundary set. */
        loop_pixel = B[B_count];
        M = 0; /* No connected neighbors yet! */
        /* Determine M neighbors, locations in neighbors */
        ConnectedNeighbors2D(loop_pixel, T, img, n_neighbors, nh_offsets, width, height, &M, neighbors);
        /* Loop through connected neighbors, add to boundary set */
        /* set seg[i][j] = ClassLabel where connected */
        for(i=0; i<M; i++){
            neighbor = neighbors[i];
            /* If neighbor hasn't been marked, it hasn't been added. */
            if(seg[neighbor.m][neighbor.n]!=ClassLabel){
                /* Mark seg as ClassLabel */
                seg[neighbor.m][neighbor.n] = ClassLabel;
                /* Add neighbor to boundary set! */
                B[B_count] = neighbor;
                /* Add to our B_count. */
                B_count += 1;
                /* increment our NumConPixels */
                (*NumConPixels)++;
            }
        }
    }
    multifree(neighbors, 1);
    multifree(B, 1);
}
```

Listing 2: connected_neighbors_sets.c

2 Section 2 Report: Image Segmentation

2.1 Print outs of the randomly colored segmentation for T = 1, T = 2, and T = 3.

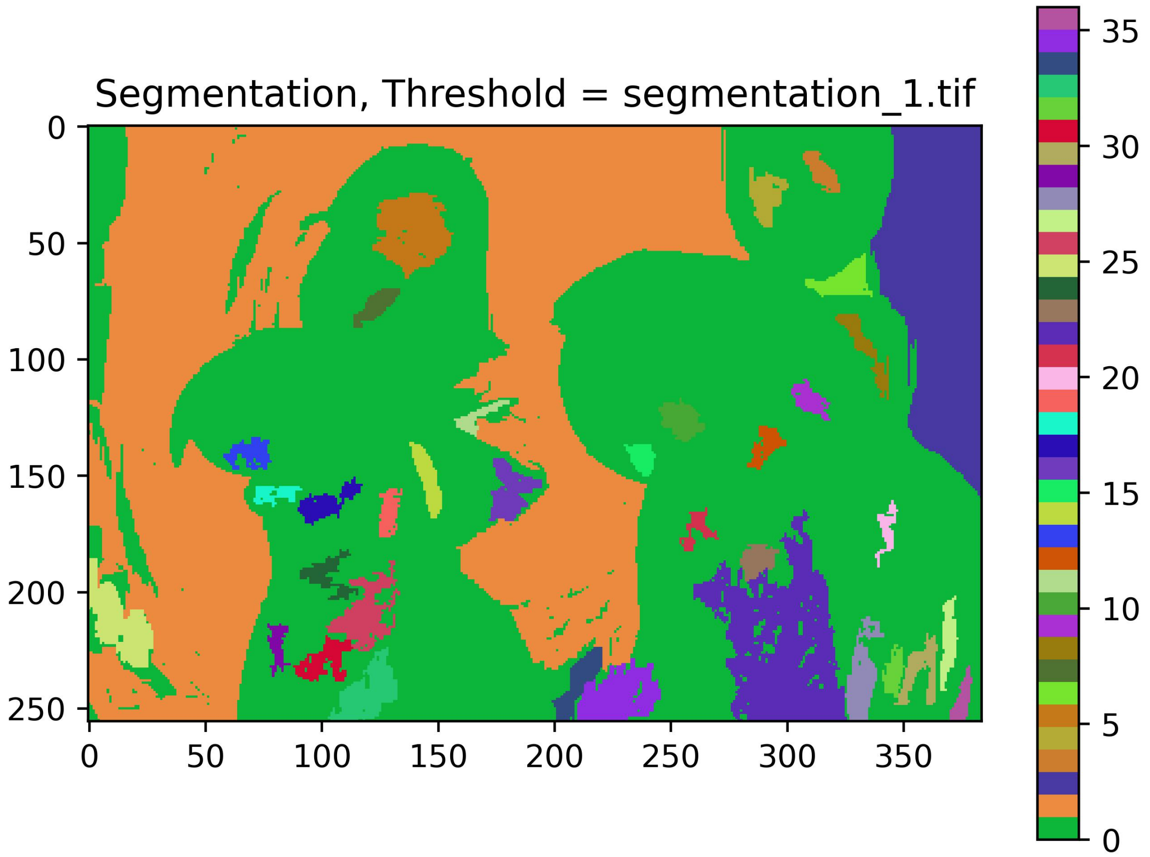

Figure 5: Randomly colored segmentation for T = 1.

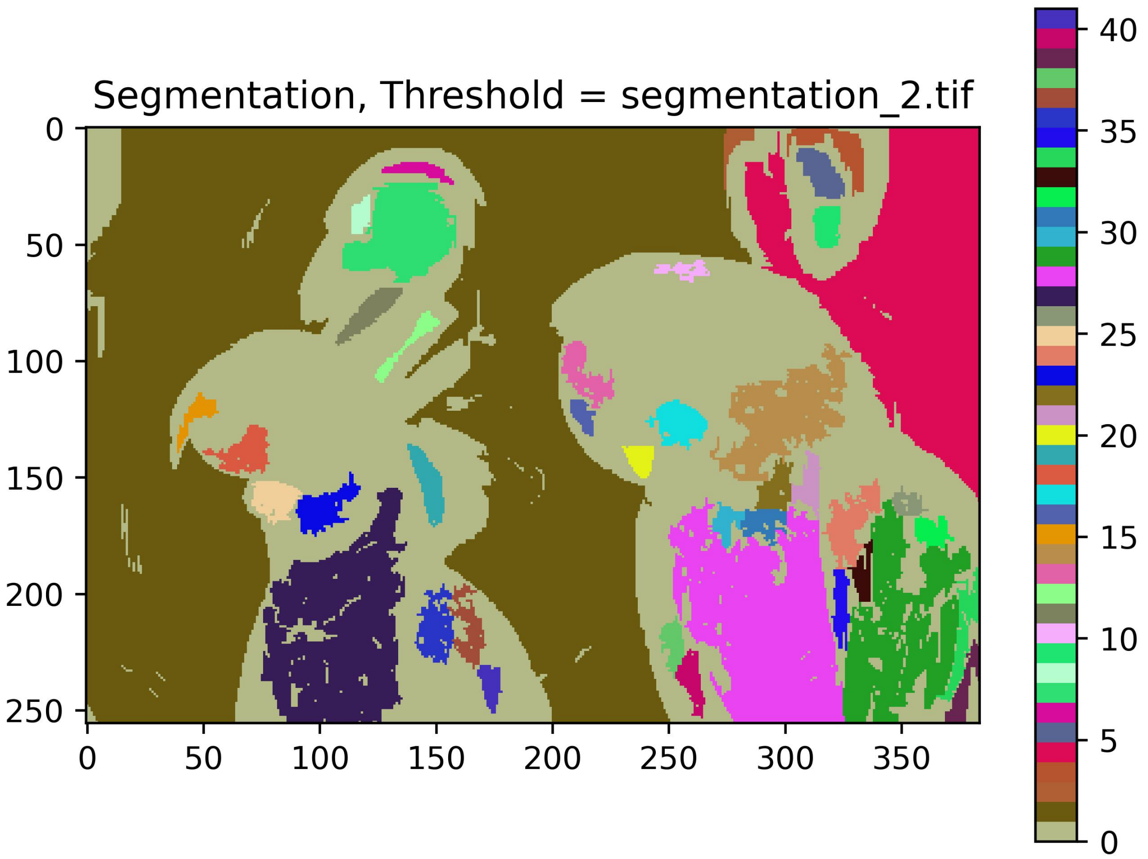

Figure 6: Randomly colored segmentation for T = 2.

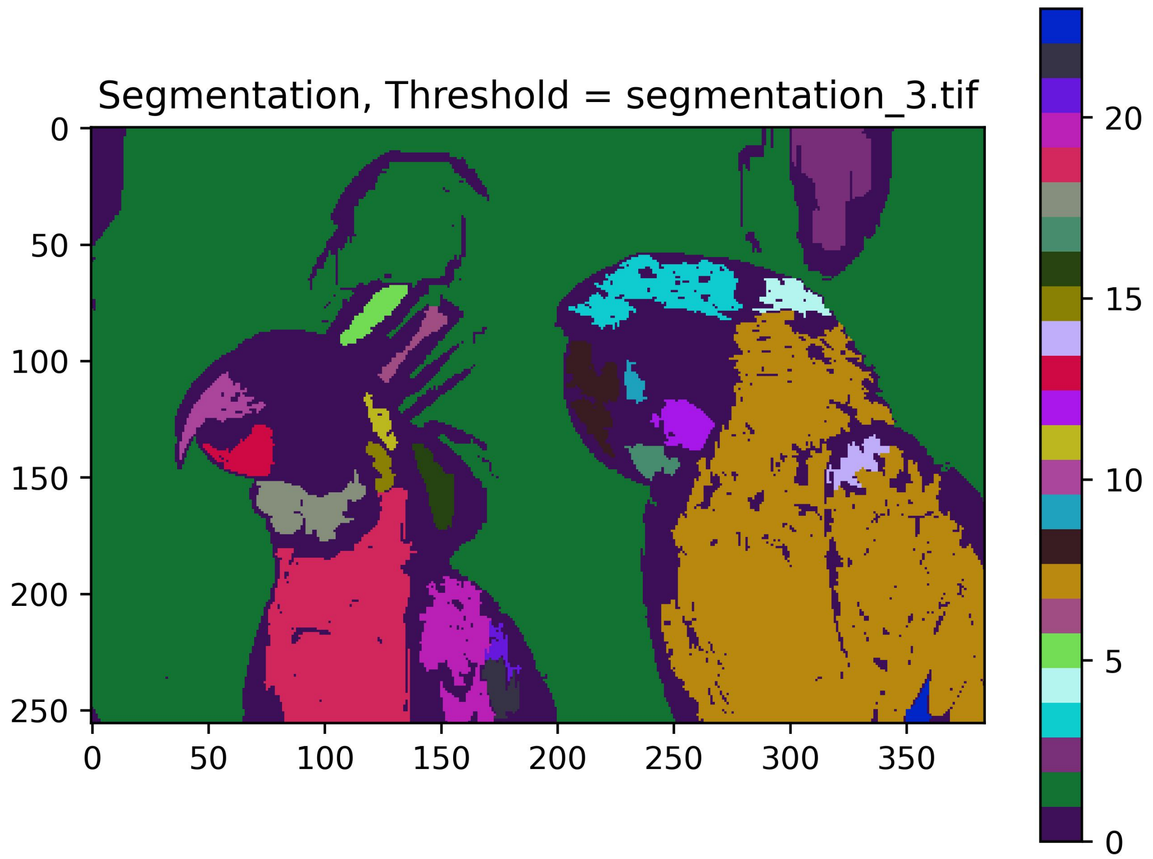

Figure 7: Randomly colored segmentation for T = 3.

2.2 A listing of the number of regions generated for each of the values of T = 1, T = 2, and T = 3.

T	Number of Regions
1	37
2	42
3	24

Table 1: The Number of Regions for each threshold. I find it particularly interesting that for a threshold of 1, there are less regions than a threshold of 2.

2.3 A listing of my C code.

It was a little tricky to implement the de-labeling until I realized that I could just find the connected set again and label it 0. Outside of that, this main function uses the same tools built for section 1.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "allocate.h"
#include "tiff.h"
#include "connected_neighbors_sets.h"

void error(char *name);

int main(int argc, char **argv){
    FILE *fp;
    struct TIFF_img input_img, seg_img;
    struct pixel seed;
    int i, j, T, ClassLabel, NumConPixels=0, NumConPixels2=0;
    unsigned int **seg;
    char outfile[100];

    /*
    3 args.
    */
    if(argc != 3) error(argv[0]);

    /* 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, &input_img)){
    fprintf(stderr, "error reading file %s\n", argv[1]);
    exit(1);
    }
    if(input_img.TIFF_type != 'g'){
    fprintf(stderr, "error:  image must be grayscale.\n");
    exit(1);
    }
    /* close image file */
    fclose(fp);

    /* Read in threshold. */
    T = atoi(argv[2]);

    /* allocate seg img populated with zeros. */
    seg = multialloc(sizeof(unsigned int), 2, input_img.height, input_img.width);
    get_TIFF(&seg_img, input_img.height, input_img.width, 'g');
    /* Initialize the connected sets to 0, as described in the algorithm */
    for(i=0; i<input_img.height; i++){
        for(j=0; j<input_img.width; j++){
            seg_img.mono[i][j] = 0;
            seg[i][j] = 0;
        }
    }
    ClassLabel = 1;
    for(i=0; i<input_img.height; i++){
        for(j=0; j<input_img.width; j++){
            /* Only if the pixel of the segmented image does not belong to a connected set */
            if(seg[i][j]==0){
                /* Set the seed in raster order. */
                seed.m = i; seed.n = j;
                NumConPixels=0;
                ConnectedSet(seed, T, input_img.mono, input_img.width, input_img.height,
                             ClassLabel, seg, &NumConPixels);
                if(NumConPixels >100){
                    /* Increment class label. */
                    ClassLabel++;
                }else{
                     NumConPixels2 = 0;
                     /* Label the small set as 0. */
                     ConnectedSet(seed, T, input_img.mono, input_img.width, input_img.height,
                                  0, seg, &NumConPixels2);
                    if(NumConPixels!=NumConPixels2){
                        error("The ConnectedSet function does not work.");
                    }
                }
            }
        }
    }
    printf("Threshold %d: %d Labels.\n",T,ClassLabel);
    for(i=0; i<input_img.height; i++){
        for(j=0; j<input_img.width; j++){
            seg_img.mono[i][j] = seg[i][j];
        }
    }

    /* Make a sensible output filename. */
    snprintf(outfile, sizeof(outfile), "images/segmentation_%d.tif", T);

        /* Try and open the file, error if can't. */
    if((fp = fopen(outfile, "wb")) == NULL){
        fprintf(stderr, "cannot open file %s\n", outfile);
        exit(1);
    }

    /* Try and write the file, error if can't. */
    if(write_TIFF(fp, &seg_img)){
        fprintf(stderr, "error writing TIFF file %s\n", outfile);
        exit(1);
    }
    /* Close the file. */
    fclose(fp);

    multifree(seg, 2);
    free_TIFF(&seg_img);
    return 0;
}

void error(char *name){
    printf("usage:  %s  image.tiff threshold\n\n",name);
    printf("this program reads in a grayscale TIFF image.\n");
    printf("It finds the connected sets from a pixel at \n");
    printf("row col for a threshold.\n");
    exit(1);
}
```
Listing 3: sec_2.c