Assignment 5

(Rajitha Bhavani Kantheti-C46686177)

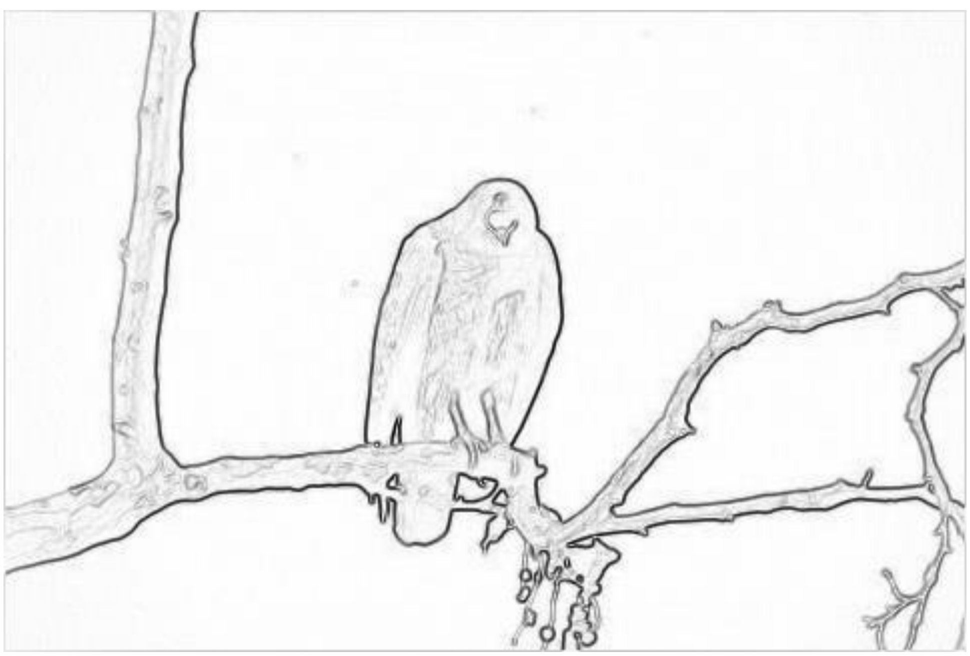
Steps for creating active contours:

1. Read gray scale PPM “hawk.ppm” image
2. Read initial contour points from “initial_contour” file.
3. Draw a “+” shape in a 7x7 window based on the initial contour points on the original PPM gray scale image.
4. Implement Active Contour Algorithm based on initial contour points.
   - The Active contour algorithm using 2 Internal Energies and 1 External Energy.
   - Each of the Energies calculated in a 7x7 window were normalized by finding the maximum and minimum value of that size window.
   - The new contour points were calculated by finding the minimum value.
5. The Active Contour algorithm was run a total of 30 iterations with the purpose of obtaining the best contour points.

IMAGE WITH INTIAL CONTOURS:

SOBEL EDGE GRADIENT MAGNITUDE IMAGE

INVERTED SOBEL EDGE GRADIENT IMAGE



FINAL IMAGE:

FINAL CONTOUR POINTS:

COLS	ROWS
266	104
272	114
275	123
277	133
278	143
278	154
275	167
270	180
266	191
262	201
255	213
254	226
246	237
235	234
225	239
226	249
221	260

212	266
199	266
195	255
194	245
185	242
177	237
185	223
180	211
181	201
182	190
183	180
184	170
186	160
188	147
193	134
196	125
199	116
206	109
214	105
222	100
230	94
237	87
246	84
256	86
264	96

CODE:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

void find_min_and_max_float(float *convolution_image, int image_rows, int image_cols, float *min,
float *max)
{
	// VARIABLE DECLARATION SECTION
	int i, j, k;

	*min = convolution_image[0];
	*max = convolution_image[0];

	for(i = 1; i < (image_rows-1); i++)
	{
		for(j = 1; j < (image_cols-1); j++)
		{
			k = (i * image_cols) + j;
			if (*min > convolution_image[k])
			{
				*min = convolution_image[k];
			}
			if (*max < convolution_image[k])
			{
				*max = convolution_image[k];
			}
		}
	}
}

float *normalize_float(float *convolution_image, int image_rows, int image_cols, float new_min, float
new_max, float min, float max)
{
	float *normalized_image;
	int i;

	normalized_image = (float *)calloc(image_rows * image_cols, sizeof(float));

	for (i = 0; i < (image_rows * image_cols); i++)
	{
```

```
		normalized_image[i] = ((convolution_image[i] - min)*(new_max - new_min)/(max-min))
+ new_min;
	}

	return normalized_image;
}

int main()
{
FILE *file;
int ROWS, COLS, BYTES;
char HEADER[320];
unsigned char *input_image;
float *sobel_image;
int *contour_rows, *contour_cols;
int file_size;

if ((file = fopen("hawk.ppm", "rb")) == NULL)
{
printf("Error, could not read input image\n");
exit(0);
}
fscanf(file, "%s %d %d %d\n", HEADER, &COLS, &ROWS, &BYTES);
if ((strcmp(HEADER, "P5") != 0) || (BYTES !=  255))
{
printf("Error, not a greyscale 8-bit PPM image\n");
exit(0);
}

input_image = (unsigned char *)calloc(ROWS * COLS, sizeof(unsigned char));
//HEADER[0]=fgetc(file);
fread(input_image, sizeof(unsigned char), ROWS * COLS, file);
fclose(file);

FILE *fpt;
int i = 0;
int cols, rows;
char c;
cols = rows = 0;
file_size = 0;

// OBTAINS FILE LENGTH AND REWINDS IT TO BEGINNING
fpt = fopen("initial_contour", "r");
if (fpt == NULL)
{
printf("Error, could not read in initial contour text file\n");
```

```
exit(1);
}
printf("hello");
while((c = fgetc(fpt)) != EOF)
{
if (c == '\n')
{
file_size += 1;
}
}
rewind(fpt);

// ALLOCATES MEMORY
contour_rows = calloc(file_size, sizeof(int *));
contour_cols = calloc(file_size, sizeof(int *));

// EXTRACTS THE INITIAL COLUMNS AND ROWS OF INITIAL CONTOUR TEXT FILE
while((fscanf(fpt, "%d %d\n", &cols, &rows)) != EOF)
{
(contour_rows)[i] = rows;
(contour_cols)[i] = cols;
i++;
}

fclose(fpt);

unsigned char *output_image;
rows=cols=0;
i = 0;

output_image = (unsigned char *)calloc(ROWS * COLS, sizeof(unsigned char));

// COPIES ORIGINAL IMAGE TO OUTPUT IMAGE
for (i = 0; i < (ROWS * COLS); i++)
{
output_image[i] = input_image[i];
}

// DRAW "+" ON IMAGE
for (i = 0; i < file_size; i++)
{
rows = (contour_rows)[i];
cols = (contour_cols)[i];

// "|" ON COLS
output_image[(rows - 3)*COLS + cols] = 0;
```

```
output_image[(rows - 2)*COLS + cols] = 0;
output_image[(rows - 1)*COLS + cols] = 0;
output_image[(rows - 0)*COLS + cols] = 0;
output_image[(rows + 1)*COLS + cols] = 0;
output_image[(rows + 2)*COLS + cols] = 0;
output_image[(rows + 3)*COLS + cols] = 0;

// "-" ON ROWS
output_image[(rows * COLS) + (cols - 3)] = 0;
output_image[(rows * COLS) + (cols - 2)] = 0;
output_image[(rows * COLS) + (cols - 1)] = 0;
output_image[(rows * COLS) + (cols - 0)] = 0;
output_image[(rows * COLS) + (cols + 1)] = 0;
output_image[(rows * COLS) + (cols + 2)] = 0;
output_image[(rows * COLS) + (cols + 3)] = 0;
}

file = fopen("hawk_initial_contour.ppm", "w");
fprintf(file, "P5 %d %d 255\n", COLS, ROWS);
fwrite(output_image, ROWS * COLS, sizeof(unsigned char), file);
fclose(file);
/* UN-NORMALIZED SOBEL IMAGE */

int *convolution_image;
unsigned char *normalized_image;
int  j;
i=rows=cols=0;
int index1 = 0;
int index2 = 0;
int x = 0;
int y = 0;
int min = 0;
int max = 0;

// X and Y KERNELS
int g_x[9] = {-1, 0, 1, -2, 0, 2, -1, 0, 1};

int g_y[9] = {-1, -2, -1, 0, 0, 0, 1, 2, 1};

// ALLOCATE MEMORY
convolution_image = (int *)calloc(ROWS * COLS, sizeof(int));
sobel_image = (float *)calloc(ROWS * COLS, sizeof(float));

// COPY ORIGINAL IMAGE
for (i = 0; i < (ROWS * COLS); i++)
{
```

```
convolution_image[i] = input_image[i];
}

// CONVOLUTE INPUT IMAGE WITH X AND Y KERNELS
for (rows = 1; rows < (ROWS - 1); rows++)
{
for (cols = 1; cols < (COLS - 1); cols++)
{
x = 0;
y = 0;
for (i = -1; i < 2; i++)
{
for (j = -1; j < 2; j++)
{
index1 = (COLS * (rows + i)) + (cols + j);
index2 = 3*(i + 1) + (j + 1);
x += (input_image[index1] * g_x[index2]);
y += (input_image[index1] * g_y[index2]);
}
}
index1 = (COLS * rows) + cols;
convolution_image[index1] = sqrt(x*x + y*y);
sobel_image[index1] = sqrt(x*x + y*y);
}
}

// FIND MINIMUM AND MAXIMUM VALUES IN CONVOLUTED IMAGE

i=0;

min = convolution_image[0];
max = convolution_image[0];
for (i = 1; i < (ROWS * COLS); i++)
{
if (min > convolution_image[i])
{
min = convolution_image[i];
}
if (max < convolution_image[i])
{
max = convolution_image[i];
}
}

// NORMALIZES CONVOLUTED IMAGE FROM RANGE OF 0-255 IN ORDER TO SAVE AS PPM
```

```
i=0;

// Allocate memory
normalized_image = (unsigned char *)calloc(ROWS * COLS, sizeof(unsigned char));

for (i = 0; i < (ROWS * COLS); i++)
{
if (min == 0 && max == 0)
{
normalized_image[i] = 0;
}
else
{
normalized_image[i] = ((convolution_image[i] - min)*(255 - 0)/(max-min)) + 0;
}
}


file = fopen("hawk_sobel_image.ppm", "w");
fprintf(file, "P5 %d %d 255\n", COLS, ROWS);
fwrite(normalized_image, ROWS * COLS, sizeof(unsigned char), file);
fclose(file);

// INVERTS NORMALIZED IMAGE IN ORDER TO SAVE AS PPM
for (i = 0; i < (ROWS * COLS); i++)
{
normalized_image[i] = 255 - normalized_image[i];
}

file = fopen("hawk_sobel_inverted_image.ppm", "w");
fprintf(file, "P5 %d %d 255\n", COLS, ROWS);
fwrite(normalized_image, ROWS * COLS, sizeof(unsigned char), file);
fclose(file);

        // FREE ALLOCATED MEMORY


        /* CONTOUR ALGORITHM */

        float *inverted_sobel;
        float *first_internal_energy;
        float *second_internal_energy;
        float *external_energy;
        float *sum_energies;
        float min, max, new_min, new_max;
```

```
	float *first_internal_energy_normalized, *second_internal_energy_normalized,
*external_energy_normalized;
	float average_distance_x = 0;
	float average_distance_y = 0;
	float average_distance = 0;
	int  k, l;
	i=j=rows= cols=0;
	int index = 0;
	int index2 = 0;
	int index3 = 0;
	int new_x[arr_length];
	int new_y[arr_length];
	int temp = 0;
	new_min = 0.0;
	new_max = 1.0;


	// ALLOCATE MEMORY
	first_internal_energy = (float *)calloc(49, sizeof(float));
	second_internal_energy = (float *)calloc(49, sizeof(float));
	external_energy = (float *)calloc(49, sizeof(float));
	sum_energies = (float *)calloc(49, sizeof(float));
	inverted_sobel = (float *)calloc(ROWS * COLS, sizeof(float));

	// FIND MINIMUM AND MAXIMUM VALUE OF SOBEL IMAGE

	i=j=k=0;
	min = convolution_image[0];
	max = convolution_image[0];

	for(i = 1; i < (ROWS-1); i++)
	{
		for(j = 1; j < (COLS-1); j++)
		{
			k = (i * COLS) + j;
			if (min > convolution_image[k])
			{
				min = convolution_image[k];
			}
			if (max < convolution_image[k])
			{
				max = convolution_image[k];
			}
		}
	}
```

```
        // Creates an invereted Sobel image for External Energy calculation
        for ( i = 0; i < (ROWS * COLS); i++)
        {
                inverted_sobel[i] = sobel_image[i];
                inverted_sobel[i] = (float)max - inverted_sobel[i];
        }

        // Calculates first Internal Energy
        for (l = 0; l < 30; l++)
        {

                average_distance_x = 0.0;
                average_distance_y = 0.0;
                average_distance = 0.0;

                // CALCULATES THE AVERAGE DISTANCE BETWEEN CONTOUR POINTS
                for (i = 0; i < file_size; i++)
                {
                        if ((i + 1) < file_size)
                        {
                                average_distance_x = (*contour_cols)[i] - (*contour_cols)[i +
1])*(*contour_cols)[i] - (*contour_cols)[i + 1]);
                                average_distance_y = (*contour_rows)[i] - (*contour_rows)[i +
1])*(*contour_rows)[i] - (*contour_rows)[i + 1]);
                        }
                        else
                        {
                                average_distance_x = (*contour_cols)[i] -
(*contour_cols)[0])*(*contour_cols)[i] - (*contour_cols)[0]);
                                average_distance_y = (*contour_rows)[i] -
(*contour_rows)[0])*(*contour_rows)[i] - (*contour_rows)[0]);
                        }
                        average_distance +=sqrt(average_distance_x + average_distance_y);
                        new_x[i] = 0;
                        new_y[i] = 0;
                }
                average_distance /= file_size;

                for (i = 0; i < file_size; i++)
                {
                        rows = (contour_rows)[i];
                        cols = (contour_cols)[i];
                        index = 0;

                        // FIRST AND SECOND INTERNAL ENERGY AND EXTERNAL ENERGY CALCULATED
                        for (j = (rows - 3); j <= (rows + 3); j++)
```

```
			{
				for (k = (cols - 3); k <= (cols + 3); k++)
				{
					if ((i + 1) < file_size)
					{
						first_internal_energy[index] = (k - (*contour_cols)[i + 1]) +
(j - (*contour_rows)[i + 1])*(j - (*contour_rows)[i + 1])*(k - (*contour_cols)[i + 1]) + (j -
(*contour_rows)[i + 1])*(j - (*contour_rows)[i + 1]);
						second_internal_energy[index] =
(sqrt(first_internal_energy[index]) - average_distance)*(sqrt(first_internal_energy[index]) -
average_distance);
						index2 = (j * COLS) + k;
						external_energy[index] =
(inverted_sobel[index2])*(inverted_sobel[index2]);
					}
					else
					{
						first_internal_energy[index] = (k - (*contour_cols)[0]) + (j -
(*contour_rows)[0])*(j - (*contour_rows)[0])*(k - (*contour_cols)[0]) + (j - (*contour_rows)[0])*(j -
(*contour_rows)[0]);
						second_internal_energy[index] =
(sqrt(first_internal_energy[index]) - average_distance)*(sqrt(first_internal_energy[index]) -
average_distance);
						index2 = (j * COLS) + k;
						external_energy[index] =
(inverted_sobel[index2])*(inverted_sobel[index2]);

					}
					index++;
				}
			}

			// FINDS MINIMUM AND MAXIMUM VALUES OF EACH ENERGY AND
NORMALIZES TO VALUE OF 0 AND 1
			find_min_and_max_float(first_internal_energy, 7, 7, &min, &max);
			first_internal_energy_normalized = normalize_float(first_internal_energy, 7, 7,
new_min, new_max, min, max);
			find_min_and_max_float(second_internal_energy, 7, 7, &min, &max);
			second_internal_energy_normalized = normalize_float(second_internal_energy,
7, 7, new_min, new_max, min, max);
			find_min_and_max_float(external_energy, 7, 7, &min, &max);
			external_energy_normalized = normalize_float(external_energy, 7, 7, new_min,
new_max, min, max);

			// CALCULATES THE ENERGY
			for (j = 0; j < 49; j++)
```

```
			{
				sum_energies[j] = first_internal_energy_normalized[j] + 
second_internal_energy_normalized[j] + external_energy_normalized[j];
			}

			// DETERMINES THE LOWEST VALUE FOR NEW POINTS
			min = sum_energies[0];
			index = 0;
			for (j = 0; j < 49; j++)
			{
				if (min > sum_energies[j])
				{
					min = sum_energies[j];
					index = j;
				}
			}

			// DETERMINES ROW AND COLUMN FOR NEW POINT BASED ON INDEX
			temp = 0;
			index2 = (index / 7); // row
			if (index2 < 3)
			{
				temp = (contour_rows)[i] - abs(index2 - 3);
				new_y[i] = temp;
			}
			else if (index2 > 3)
			{
				temp = (contour_rows)[i] + abs(index2 - 3);
				new_y[i]  = temp;
			}
			else
			{
				new_y[i] = (contour_rows)[i];
			}


			index3 = (index % 7); // col
			if (index3 < 3)
			{
				new_x[i] = (contour_cols)[i] - abs(index3 - 3);

			}
			else if (index3 > 3)
			{
				new_x[i] = (contour_cols)[i] + abs(index3 - 3);
			}
```

```
                else
                {
                        new_x[i] = (contour_cols)[i];
                }
        }

        // SETS NEW POINTS
        for (i = 0; i < file_size; i++)
        {
                (contour_cols)[i] = new_x[i];
                (contour_rows)[i] = new_y[i];
        }

}

// DRAWS CONTOUR WITH FINAL POINTS

rows= cols=0;
i = 0;

output_image = (unsigned char *)calloc(ROWS * COLS, sizeof(unsigned char));

// COPIES ORIGINAL IMAGE TO OUTPUT IMAGE
for (i = 0; i < (ROWS * COLS); i++)
{
        output_image[i] = input_image[i];
}

// DRAW "+" ON IMAGE
for (i = 0; i < file_size; i++)
{
        rows = (contour_rows)[i];
        cols = (contour_cols)[i];

        // "|" ON COLS
        output_image[(rows - 3)*COLS + cols] = 0;
        output_image[(rows - 2)*COLS + cols] = 0;
        output_image[(rows - 1)*COLS + cols] = 0;
        output_image[(rows - 0)*COLS + cols] = 0;
        output_image[(rows + 1)*COLS + cols] = 0;
        output_image[(rows + 2)*COLS + cols] = 0;
        output_image[(rows + 3)*COLS + cols] = 0;

        // "-" ON ROWS
        output_image[(rows * COLS) + (cols - 3)] = 0;
        output_image[(rows * COLS) + (cols - 2)] = 0;
```

```
            output_image[(rows * COLS) + (cols - 1)] = 0;
            output_image[(rows * COLS) + (cols - 0)] = 0;
            output_image[(rows * COLS) + (cols + 1)] = 0;
            output_image[(rows * COLS) + (cols + 2)] = 0;
            output_image[(rows * COLS) + (cols + 3)] = 0;
        }

        // SAVES IMAGE WITH CONTOUR POINTS EXPRESSED AS "+"

        file = fopen("hawk_final_contour.ppm", "w");
        fprintf(file, "P5 %d %d 255\n", COLS, ROWS);
        fwrite(output_image, ROWS * COLS, sizeof(unsigned char), file);
        fclose(file);

        // CREATE FILE WITH FINAL CONTOUR POINTS
        FILE *file;
        file = fopen("final_contour_points.csv", "w");
        fprintf(file, "COLS,ROWS\n");
        for(i = 0; i < file_size; i++)
        {
            fprintf(file, "%d,%d\n", (contour_cols)[i], (contour_rows)[i]);
        }
        fclose(file);


        return 0;
}
```