# BIRD COUNTING

The main purpose of this study is providing an algorithm that can count birds for all these three images.

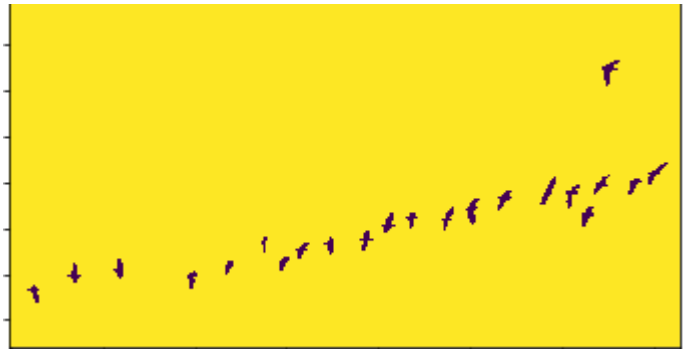
Number of birds: 10

Number of birds: 3

Number of birds: 22

Although it may seem like a simple counting problem, when the segmentation is

applied directly, the following results will be observed so counting will fail.



In order to make the birds more prominent, we have to do region growing.

Thus, it will be possible to count the birds in each picture.

## REGION GROWING

Region growing is a simple region-based image segmentation method.

It is also classified as a pixel-based image segmentation method since it involves the selection of initial seed points.

This approach to segmentation examines neighboring pixels of initial seed

points and determines whether the pixel neighbors should be added to the region.

The main goal of segmentation is to partition an image into regions.

Some segmentation methods such as thresholding achieve this goal by looking for the boundaries between regions based on discontinuities in grayscale or color properties. Region-based segmentation is a technique for determining the region directly. The basic formulation is:

$(a)\ \bigcup_{i=1}^{n} R_i = R.$

$(b)\ R_i$ is a connected region, $\mathrm{i} = 1,\ 2,\ \ldots, \mathrm{n}$

$(c)\ R_i \bigcap R_j = \varnothing, i \neq j$

$(d)\ P(R_i) = \mathrm{TRUE}$ for $i = 1, 2, \ldots, n.$

$(e)\ P(R_i \bigcup R_j) = \mathrm{FALSE}$ for any adjacent region $R_i$ and $R_j$.

$P(R_i)$ is a logical predicate defined over the points in set $R_i$ and $\varnothing$ is the null set.

(a) means that the segmentation must be complete; that is, every pixel must be in a region.

(b) requires that points in a region must be connected in some predefined sense.

(c) indicates that the regions must be disjoint.

(d) deals with the properties that must be satisfied by the pixels in a segmented region. For example, $P(R_i) = \mathrm{TI}$

(e) indicates that region $R_i$ and $R_j$ are different in the sense of predicate $P$.

To implement Region Grow segmentation method there are two classes which named are "Basic.py" and "RegionGrow.py". *Basic.py* includes very basic list operations. Region growing algorithms was implemented i the *RegionGrow.py*

```
class Basic():
    def __init__(self):
        self.item = []

    def push(self, value):
        self.item.append(value)

    def pop(self):
        return self.item.pop()

    def size(self):
        return len(self.item)

    def checkEmpty(self):
        return self.size() == 0
```

*RegionGrow.py* includes 3 main functions which are *getNeighbour()* , *ApplyRegionGrow()* and *BreadthFirstSearch()*.

*getNeighbour()* is a helper function to assist to implement *ApplyRegionGrow()* function.

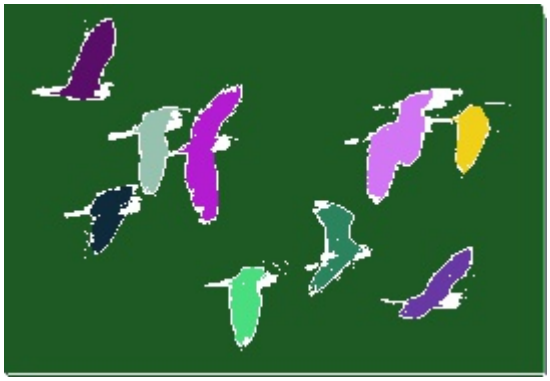
```
def getNeighbour(self, x_, y_):

        neighbour = []

        for i in (-1 ,0, 1):
            for j in (-1, 0, 1):
                if (i, j) == (0, 0):
                    continue
                x = x_ + i
                y = y_ + j
                if self.border(x, y):
                    neighbour.append((x, y))
        return neighbour
```

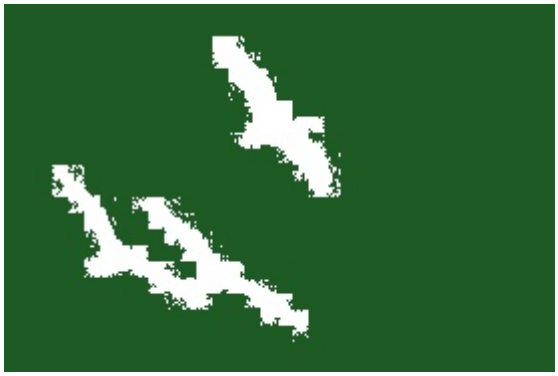
In *BreadthFirstSeach()* function implemented *Breadth First Search algorithm*.
It presents efficient algorithms for eroding, dilating, skeletonizing, and distance-transforming regions. These algorithms work by traversing regions in a breadth-first manner using a queue for storage of unprocessed pixels.

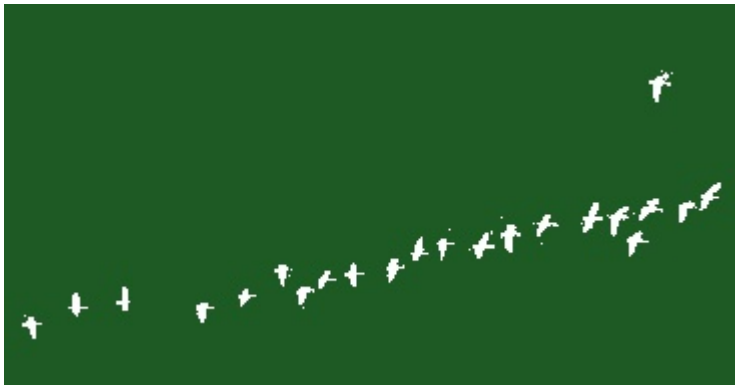
```
def BreadthFirstSearch(self, x_, y_):

        regionNum = self.changed[x_, y_]
        itemlist=[]
        itemlist.append((int(self.im[x_, y_, 0]) + int(self.im[x_, y_, 1]) +
                         int(self.im[x_,y_,2]))/3)
        var = self.thresh
        neighbours = self.getNeighbour(x_,y_)

        for x,y in neighbours:
            if self.changed[x,y] == 0 and self.distance(x, y, x_, y_) < var:
                if(self.endChange()):
                    break;
                self.changed[x, y] = regionNum
                self.heap.push((x, y))
                itemlist.append((int(self.im[x, y, 0])+int(self.im[x, y, 1]) +
                                 int(self.im[x, y, 2])) / 3)
                var = np.var(itemlist)
            var = max(var, self.thresh)
```

*ApplyRegionGrow* is implementation of region growing algorithm.
The algorithm combines the distance between the 3 color spaces ( RGB ) to

measure the homogeneity of 2 pixels. Thus, it can correctly separate the regions that have the same properties. Also, it can provide the original images which have clear edges with good segmentation results like below.







```
def ApplyRegionGrow(self):

        randomseeds = [[self.height / 2, self.width / 2],
            [self.height / 3, self.width / 3],
            [2 * self.height / 3, self.width / 3],
            [self.height/3-10,self.width/3],
            [self.height / 3, 2 * self.width / 3],
             [2 * self.height / 3, 2 * self.width / 3],
             [self.height / 3 - 10, 2 * self.width / 3],
            [self.height /3, self.width - 10],
             [2 * self.height / 3, self.width - 10],
              [self.height / 3 - 10, self.width - 10]
                    ]
```

```
        np.random.shuffle(randomseeds)

        for x_ in range (self.height):
            for y_ in range (self.width):

                if self.changed[x_,y_] == 0 and (int(self.im[x_, y_, 0]) * int(self.im[x_,

               #* int(self.im[x_, y_, 1]) * int(self.im[x_, y_, 2]) > 0) : **
                  #  **: written in case it does not appear in the report
                    self.currentRegion += 1
                    self.changed[x_,y_] = self.currentRegion
                    self.heap.push((x_,y_))
                    while not self.heap.checkEmpty():
                        x,y = self.heap.pop()
                        self.BreadthFirstSearch(x, y)
                        self.iterations += 1
                    if(self.endChange()):
                        break
                    count = np.count_nonzero(self.changed == self.currentRegion)
                    if(count < 8 * 8):
                        self.changed[self.changed == self.currentRegion]=0
                        x_-=1
                        y_-=1
                        self.currentRegion-=1

        for i in range(0, self.height):
            for j in range (0, self.width):
                val = self.changed[i][j]
                if(val == 0):
                    self.segmentation_s[i][j] = 255, 255, 255
                else:
                    self.segmentation_s[i][j] =val * 35, val * 90, val * 30

        if(self.iterations > 200000):
            print("Max Iterations")
        #print("Iterations : " + str(self.iterations))

        return  self.segmentation_s
```

The complete code for *RegionGrow.py* class is given below.

```
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 29 13:53:22 2019

@author: elifs
"""
import cv2
import numpy as np
import Basic

class regionGrow():

    def __init__(self, image_path, th_value):
```

```
    self.readImage(image_path)
    self.height, self.width,_ =  self.im.shape
    self.changed = np.zeros((self.height, self.width), np.double)
    self.currentRegion = 0
    self.iterations = 0

    #segmentation shape
    self.segmentation_s = np.zeros((self.height, self.width,3),
                                   dtype='uint8')

    self.heap = Basic.Basic()
    self.thresh = float(th_value)


def readImage(self, img_path):
    self.im = cv2.imread(img_path, 1)


def getNeighbour(self, x_, y_):

    neighbour = []

    for i in (-1 ,0, 1):
        for j in (-1, 0, 1):
            if (i, j) == (0, 0):
                continue
            x = x_ + i
            y = y_ + j
            if self.border(x, y):
                neighbour.append((x, y))
    return neighbour


def ApplyRegionGrow(self):

    randomseeds=[[self.height / 2, self.width / 2],
        [self.height / 3, self.width / 3],
        [2 * self.height / 3, self.width / 3],
         [self.height/3-10,self.width/3],
        [self.height / 3, 2 * self.width / 3],
         [2 * self.height / 3, 2 * self.width / 3],
         [self.height / 3 - 10, 2 * self.width / 3],
        [self.height /3, self.width - 10],
         [2 * self.height / 3, self.width - 10],
          [self.height / 3 - 10, self.width - 10]
                ]
    np.random.shuffle(randomseeds)

    for x_ in range (self.height):
        for y_ in range (self.width):

            if self.changed[x_,y_] == 0 and (int(self.im[x_, y_, 0]) * int(self.im[x_,
           #* int(self.im[x_, y_, 1]) * int(self.im[x_, y_, 2]) > 0) : **
```

```
                #  **: written in case it does not appear in the report

                self.currentRegion += 1
                self.changed[x_,y_] = self.currentRegion
                self.heap.push((x_,y_))
                while not self.heap.checkEmpty():
                    x,y = self.heap.pop()
                    self.BreadthFirstSearch(x, y)
                    self.iterations += 1
                if(self.endChange()):
                    break
                count = np.count_nonzero(self.changed == self.currentRegion)
                if(count < 8 * 8):
                    self.changed[self.changed == self.currentRegion]=0
                    x_-=1
                    y_-=1
                    self.currentRegion-=1

    for i in range(0, self.height):
        for j in range (0, self.width):
            val = self.changed[i][j]
            if(val == 0):
                self.segmentation_s[i][j] = 255, 255, 255
            else:
                self.segmentation_s[i][j] =val * 35, val * 90, val * 30

    if(self.iterations > 200000):
        print("Max Iterations")
    #print("Iterations : " + str(self.iterations))

    return  self.segmentation_s

    #breadth-first search algorithm
def BreadthFirstSearch(self, x_, y_):

    regionNum = self.changed[x_, y_]
    itemlist=[]
    itemlist.append((int(self.im[x_, y_, 0]) + int(self.im[x_, y_, 1]) +
                     int(self.im[x_,y_,2]))/3)
    var = self.thresh
    neighbours = self.getNeighbour(x_,y_)

    for x,y in neighbours:
        if self.changed[x,y] == 0 and self.distance(x, y, x_, y_) < var:
            if(self.endChange()):
                break;
            self.changed[x, y] = regionNum
            self.heap.push((x, y))
            itemlist.append((int(self.im[x, y, 0]) + int(self.im[x, y, 1]) + int(self.
            # + int(self.im[x, y, 2])) / 3)  **
              #  **: written in case it does not appear in the report
            var = np.var(itemlist)
        var = max(var, self.thresh)
```

```
    def endChange(self):

        return self.iterations > 200000 or np.count_nonzero(self.changed > 0) == self.widt

        #or np.count_nonzero(self.changed > 0) == self.width * self.height **
          #  **: written in case it does not appear in the report

    def border(self, x,y):
        return  0 <= x < self.height and 0 <= y < self.width

    def distance(self, x, y, x_, y_):
        return ((int(self.im[x, y, 0]) - int(self.im[x_, y_, 0])) ** 2 +
                (int(self.im[x, y, 1]) - int(self.im[x_, y_, 1])) ** 2 +
                (int(self.im[x, y, 2]) - int(self.im[x_, y_, 2])) ** 2) ** 0.5

  #  **: written in case it does not appear in the report
```

## COUNTING

Like most image analysis problems, the images need to be pre-processed.

These pre-processing steps are filtering image to reduce noise and converting

image to gray scale image.

```
def preprocess(image_path):
    image = cv2.imread(image_path)
    shifted = cv2.pyrMeanShiftFiltering(image, 21, 51)
    image = cv2.GaussianBlur(image,(5,5),0)
    gray = cv2.cvtColor(shifted, cv2.COLOR_BGR2GRAY)
    thresholding = cv2.threshold(gray, 0, 255,cv2.THRESH_BINARY | cv2.THRESH_OTSU)[1]


    #cv2.threshold(gray, 0, 255,cv2.THRESH_BINARY | cv2.THRESH_OTSU)[1] **
      #  **: written in case it does not appear in the report

    return thresholding
```

In mathematics, the Euclidean distance or Euclidean metric is the "ordinary"

straight-line distance between two points in Euclidean space. With this

distance, Euclidean space becomes a metric space.

It is neccessary to compute the exact Euclidean distance from every binary

pixel to the nearest zero pixel, then find peaks in this distance map.

Thus it can perform a connected component analysis on the local peaks,

using 8-connectivity, then Watershed algorithm can be applied.

```
def counting(pp_image, minDistance):

    EuclidianD = ndimage.distance_transform_edt(pp_image)
    localMax = peak_local_max(EuclidianD, indices = False,
                              min_distance = minDistance,labels = pp_image)
    markers = ndimage.label(localMax, structure = np.ones((3, 3)))[0]
    labels = watershed(-EuclidianD, markers, mask = pp_image)
    count = len(np.unique(labels)) - 1
    return count
```

```
def count_birds(image_path, min_distance):

    threshoulding = preprocess (image_path)
    count = counting(threshoulding, min_distance)

    return count
```

The complete code for *counting.py* class is given below.

```
from skimage.feature import peak_local_max
from skimage.morphology import watershed
from scipy import ndimage

import cv2
import numpy as np


def preprocess(image_path):
    image = image_path
    shifted = cv2.pyrMeanShiftFiltering(image, 21, 51)
    image = cv2.GaussianBlur(image,(5,5),0)
    gray = cv2.cvtColor(shifted, cv2.COLOR_BGR2GRAY)
    thresholding = cv2.threshold(gray, 0, 255,cv2.THRESH_BINARY | cv2.THRESH_OTSU)[1]

    #cv2.threshold(gray, 0, 255,cv2.THRESH_BINARY | cv2.THRESH_OTSU)[1]
    return thresholding

def counting(pp_image, minDistance):

    D = ndimage.distance_transform_edt(pp_image)
    localMax = peak_local_max(D, indices = False,
                              min_distance = minDistance,labels = pp_image)
    markers = ndimage.label(localMax, structure = np.ones((3, 3)))[0]
    labels = watershed(-D, markers, mask = pp_image)
    count = len(np.unique(labels)) - 1
    return count

def count_birds(image_path, min_distance):

    threshoulding = preprocess (image_path)
    count = counting(threshoulding, min_distance)
```

```
    return count

#  **: written in case it does not appear in the report
```

Finally *Test.py* contains test code which runs *count_birds()* function after implemented region growing. *Test.py* can count birds correctly in each image by using right threshoulding value and minimum distance value.

NOTE: Image names were changed because of space character problem:

*bird 1* changed to *1*

*bird 2* changed to *2*

*bird 3* changed to *3*

```
import RegionGrow
import counting

orj_counts = [10, 3, 22]

image1 = RegionGrow.regionGrow("bird_images/1.jpg",20)
first_image = image1.ApplyRegionGrow()

image2 = RegionGrow.regionGrow("bird_images/2.jpg",5)
second_image = image2.ApplyRegionGrow()

image3 = RegionGrow.regionGrow("bird_images/3.bmp",25)
third_image = image3.ApplyRegionGrow()


count1 = counting.count_birds(first_image, 24)
count2 = counting.count_birds(second_image, 24)
count3 = counting.count_birds(third_image, 14)

count = [count1, count2, count3]

print("Results:")
for c in range(len(count)):
    print("Number of Birds: {:2d} and Counted Birds: {:2d}".format(count[c],
                                                                  orj_counts[c]))
```

Output:

```
In [9]: runfile('C:/Users/elifs/Test.py', wdir='C:/Users/elifs')
Reloaded modules: RegionGrow, Basic
Results:
Number of Birds: 10 and Counted Birds: 10
Number of Birds:  3 and Counted Birds:  3
Number of Birds: 22 and Counted Birds: 22
```

## REFERENCES


- https://stackoverflow.com/questions/5851266/region-growing-algorithm
- https://www.wikizeroo.org/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kv(
- https://www.edureka.co/blog/breadth-first-search-algorithm/
- https://users.cs.cf.ac.uk/Dave.Marshall/Vision_lecture/node35.html
- https://en.wikipedia.org/wiki/Euclidean_distance
- https://github.com/LiChuanXOfSJTU/RegionGrowin
- https://lengrand.fr/simple-region-growing-implementation-in-python/
- https://scikit-image.org/docs/dev/auto_examples/segmentation/plot_watershed.html