# Explanation and Implementation of Snowflake Boxing Algorithm in Python

Igor Ryzhkov

October 27, 2015

## Contents

**1 Introduction** **1**

**2 Explanation and Implementation** **2**

2.1 BFS - explanation . . . . . . . . . . 2

2.2 BFS - implementation . . . . . . . . . . 3

2.3 Algorithm all together . . . . . . . . . . 4

**3 Results** **5**

**4 Conclusion** **8**

**Abstract**

The final goal of ECE311 Honors project is to write software, that will classify snowflakes in the photos taken by multi-angle snowflake camera (MASC). Our first assignment was to box the snowflackes from the images. After some time, I came up with the algorithm described in this paper, to box snowflakes automaticaly. The idea behind the algorithm is really simple: find individual snowflakes (connected components), and cut them out.

## 1 Introduction

Finding and recognizing objects in the image is a difficult problem in computer science. However, in this case it is not that hard. Because all of the images look like one in Figure 1, it is not that hard to separate snowflake from the background.

The hardest part of the problem is to find boundaries for each individual snowflake. In order to solve this problem, I use breadth-first search algorithm (BFS), with some extra return variables. Running several BFS on the pictures allows us to calculate the number of individual snowflakes, and those extra return variables can give us the boundaries for each snowflake.

That is basically all there is to my algorithm.

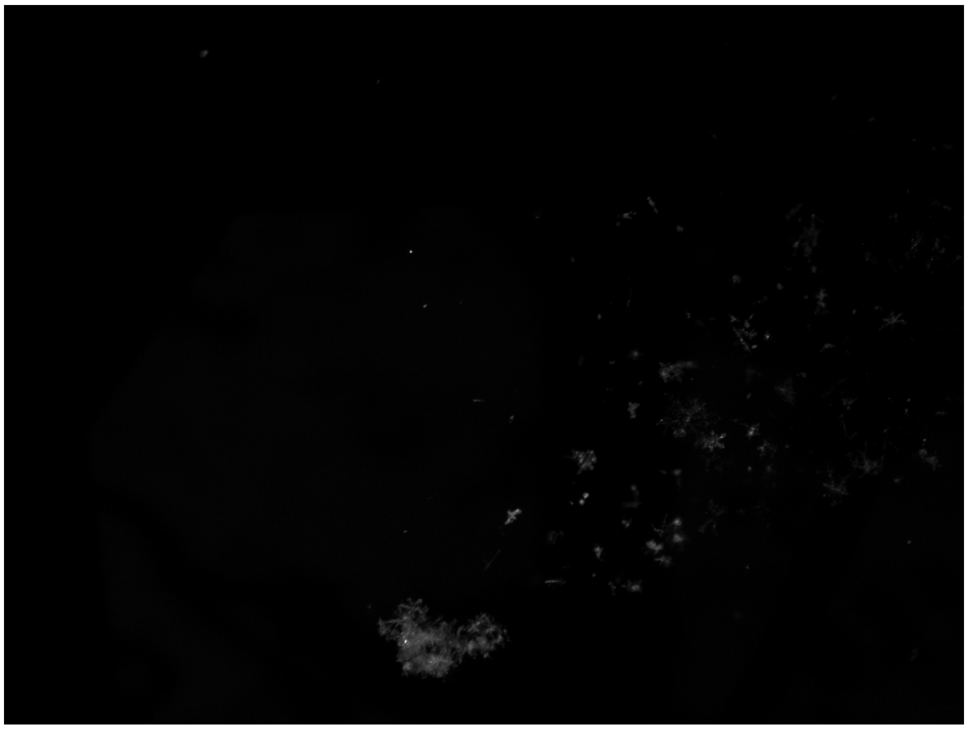

Figure 1: Example of a photo made by MASC

## 2 Explanation and Implementation

In this section I will explain in more detail how and why the program works with examples of Python code.

### 2.1 BFS - explanation

BFS is a graph algorithm. It allows us to learn the minimal number of edges that we need to walk through in order to get from source vertex to the given one. Another use of it is to learn what are the connected components of the graph.

Here is how it works. We assign $result = +\infty$ for every vertex in the graph except for the source. $resultofsource = 0$. We create a que with one element, source vertex. After that do the following:

```
while (que is not empty):
    v = que.pop()

    for each neighbour of v:
        if neighbour.result > v.result+1:
            neighbour.result = v.result+1
            que.append(neighbour)
```

## 2.2 BFS - implementation

After a few changes to fit our problem, I came up with the following code for the BFS:

```
def bfs (data, mask, i, j):
    iMax, iMin = i,i
    jMax, jMin = j,j

    q = [(i,j)]
    mask[i,j] = 1

    h, l = data.shape

    while (len(q) > 0):
        i, j = q.pop(0)
        if (i > iMax):
            iMax = i
        if (i < iMin):
            iMin = i
        if (j > jMax):
            jMax = j
        if (j < jMin):
            jMin = j

        if (i+1 < h):
            if (data[i+1,j] == 255 and mask[i+1,j] == 0):
                mask[i+1,j] = 1
                q.append((i+1,j))
        if (j+1 < l):
            if (data[i,j+1] == 255 and mask[i,j+1] == 0):
                mask[i,j+1] = 1
                q.append((i,j+1))
        if (i-1 >= 0):
            if (data[i-1,j] == 255 and mask[i-1,j] == 0):
                mask[i-1,j] = 1
                q.append((i-1,j))
        if (j-1 >= 0):
            if (data[i,j-1] == 255 and mask[i,j-1] == 0):
                mask[i,j-1] = 1
                q.append((i,j-1))

    box = (iMax, iMin, jMax, jMin)

    return mask, box
```

As you can see, this BFS returns the boundaries of the snowflake.

## 2.3 Algorithm all together

Now, that we have BFS, I can explain how full algorithm works. Here is python code for it:

```
def boxImage (filename, minDim = 30, t=24, savePath=None):

    img = imread(filename)
    helpImg = np.zeros(img.shape)

    h, l = img.shape

    hm, lm = int(h*0.95),int(l*0.85)
    hM, lM = int(h*0.05),int(l*0.15)

    im, jm, iM, jM = hm, lm, hM, lm

    for r in range(hM,hm):
        for c in range(lM,lm):
            if img[r,c] > t:
                if (r+2 > iM):
                    iM = r+2
                if (r-2 < im):
                    im = r-2
                if (c+2 > jM):
                    jM = c+2
                if (c-2 < jm):
                    jm = c-2
                helpImg[r-2:r+2,c-2:c+2] = 255
```

In this part of the code, the algorithm loads the image, and then saturates the pixel and its neighbours' values, if they are bigger then threshold t.

```
    mask = np.zeros(img.shape)
    boxes = []

    for i in range(im,iM,5):
        for j in range(jm,jM,5):
            if (helpImg[i,j] == 255 and mask[i,j] == 0):
                mask, box = bfs(helpImg, mask, i, j)
                boxes.append(box)
```

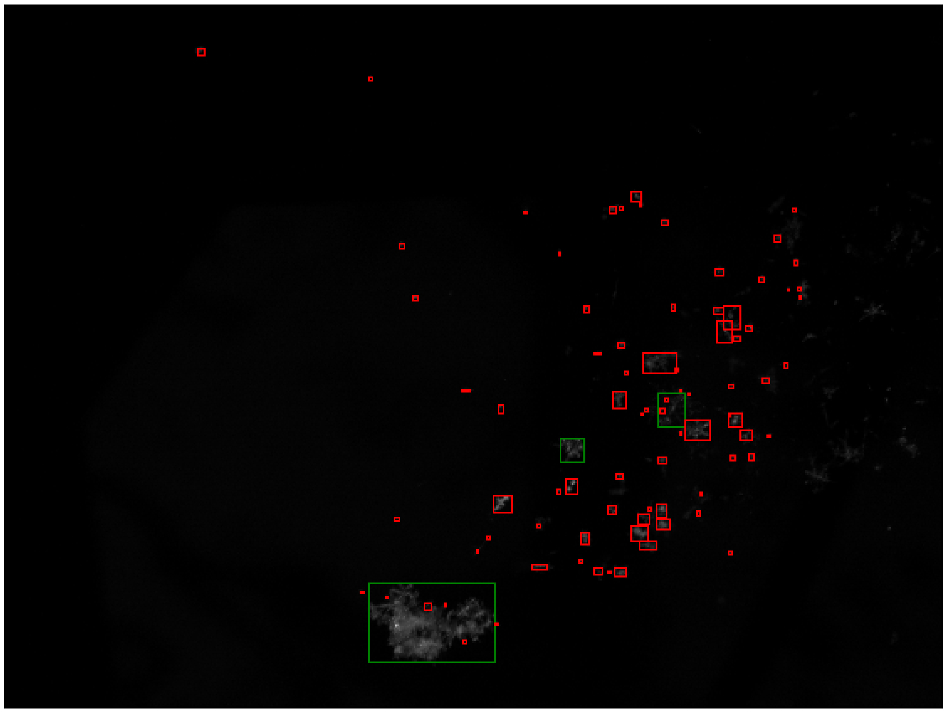

Figure 2: Demonstration of the Algorithm

In this part of the code, the algorithms runs BFS on every pixel of image that is saturated and has never been visited by BFS before. This gives us the boundaries of each individual snowflake, so we store them in the list of boundaries.

```
images = []

for box in boxes:
    yM, ym, xM, xm = box
    if (yM - ym >= minDim and xM - xm >= minDim):
        images.append(img[ym:yM, xm:xM])

return images
```

This is the last part of the code, it checks, if the image is big enough to make sense out of it, and saves it if it is.

## 3 Results

In this section I am going to provide results for the program.

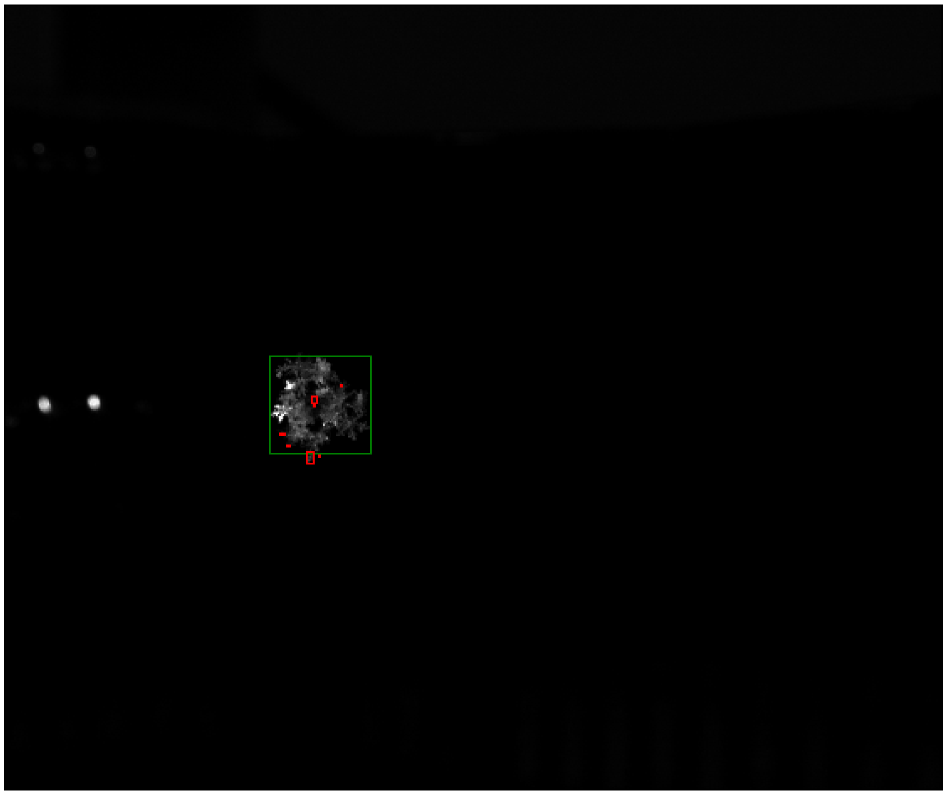

Figure 3: Demonstration of the Algorithm

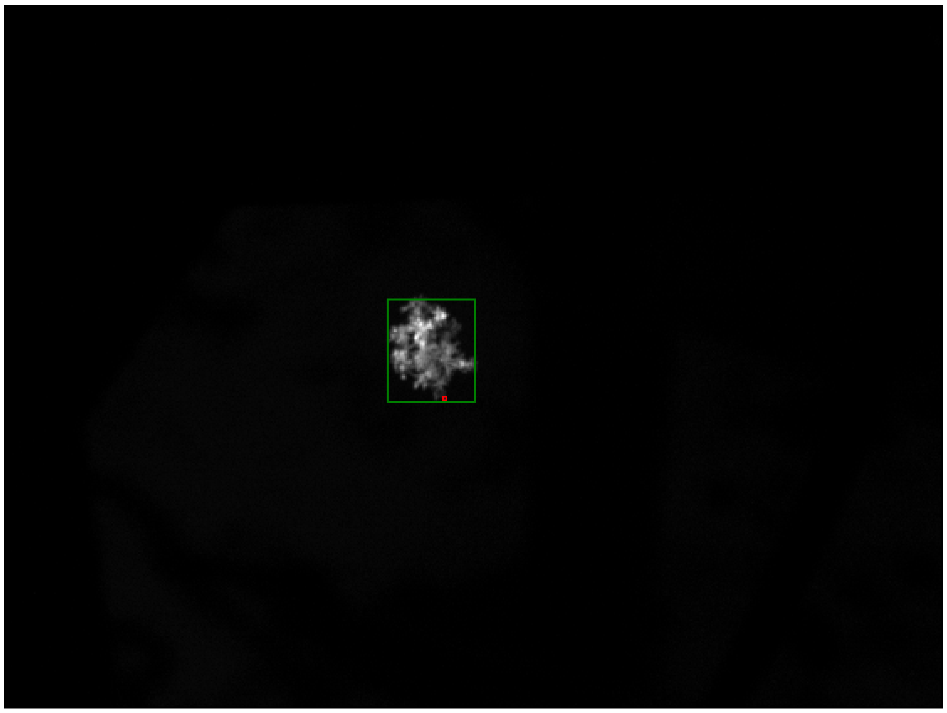

Figure 4: Demonstration of the Algorithm

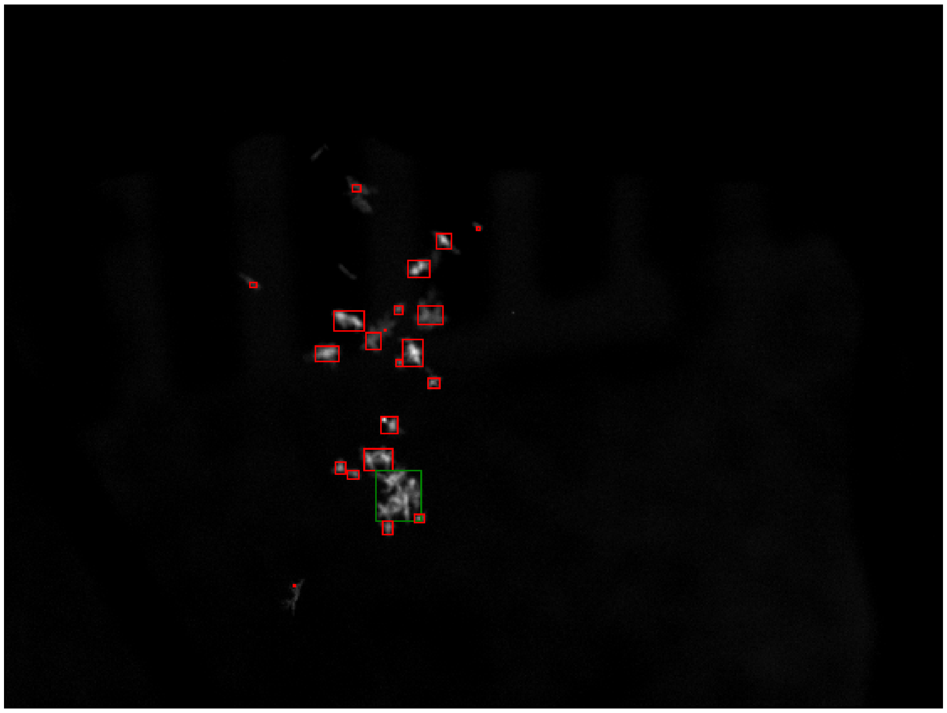

Figure 5: Demonstration of the Algorithm

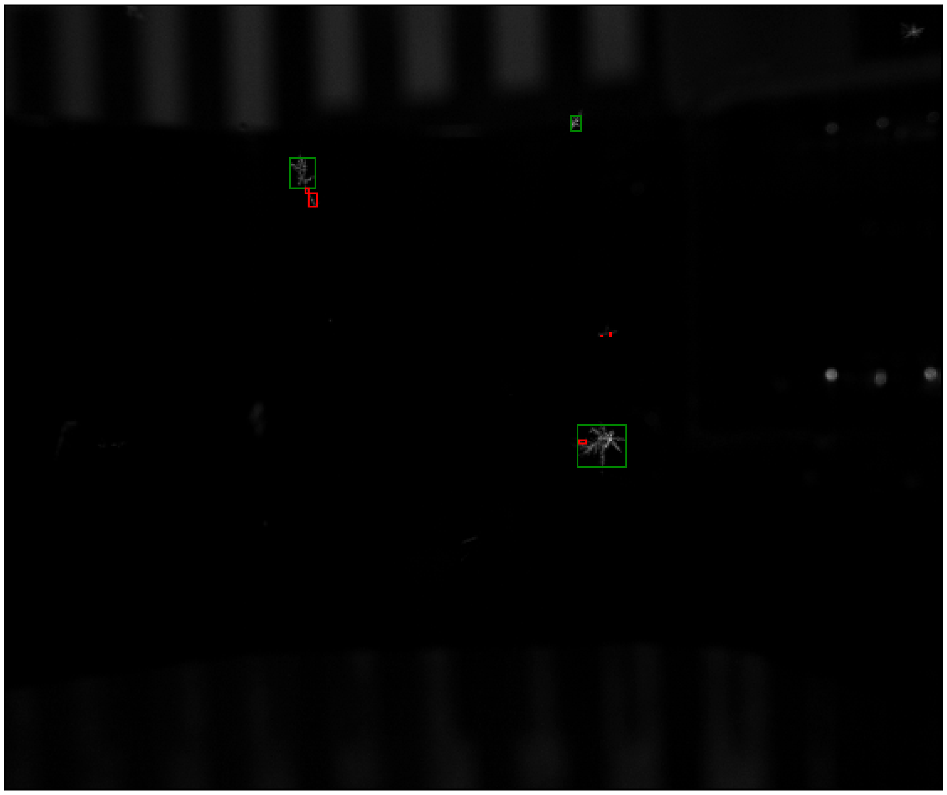

Figure 6: Demonstration of the Algorithm

## 4 Conclusion

In this paper I presented and explained (kind of) the algorithm that I have developed for the ECE311 Honors project.

This is not final version of the paper.