MEF University

Computer Engineering

Programming Studio

Object Counting With Levialdi And TSF Algorithms

Ezgi Subaşı

Muhittin Gokmen

05.03.2019

Abstract:

In order to increase the performance of electronic devices and to accelerate the operations, and speeding up the iterations while counting the objects, Parallel Shrinking Algorithms are necessary. In this article, Levialdi's Algorithm and TSF Algorithm are observed and tested to understanding their algorithms' logic and compared for usability.

Problem Definition:

Counting objects is a challenging and prevalently used image processing task in many industrial fields. Getting a certain number of elements is based for object counting. These founded elements' numbers can be used as a source of information for quantitative analysis, motion tracking, and qualitative analysis which will be useful in industries, research institutes, laboratories and so on. [1]

While our generation surrounded by technology, it is logical that wanting development and innovation from this technology. High performance and speed come at the forefront of these expected innovations. Parallel algorithms are necessary for implementing personal computers to high-performance supercomputers. Parallel components can be provided in all electronic devices such as computing nodes can consist of several processors, processors have numerous cores, each core has several independent functional units that can be pipelined as well. With parallel work of hardware and software's, performance and speed can be improved. [2]

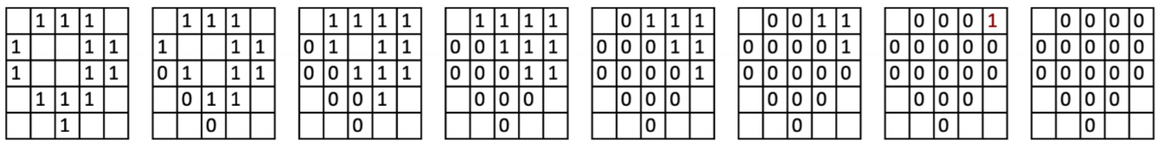
Solution Methods:

1. Levialdi Algorithm:

Levialdi's parallel algorithm has been developed in order to obtain isolated binary pixel with shrinking on each pattern and counting the number of objects with this method. This procedure can be used for counting patterns on a matrix, and a hardware implementation of the algorithm using large scale integration.

Firstly, finding the isolated point will be provided by considering about 8-neighbors of the center point. If all of the 8-neighbors are zero and the main pixel is 1, it'll be erased it and object number increased.

The algorithm continues with two specific conditions which are deletion and augmentation. While changing 1 to 0 or 0 to 1, every iterations' process with a 2x2 neighborhood. To occur deletion condition which is a translation of 1 to 0 needs a left, down and left-cross neighbors as zero and the pixel itself must be 1. For the augmentation condition, it is sufficient to have left and down neighbors as 1, the pixel is converted from 0 to 1. [3]



2. TSF Algorithm:

TSF algorithm inspired by the pattern of a checkerboard to using two-subfields. Unlike Levialdi, it is an algorithm which makes parallel shrinking with multi-directionality. Multidirectional reducing provides iterations' numbers decrease which occurs the counting of the object's number faster. According to artificial and real image tests, this algorithm proved its speed and efficiency.

In TSF algorithm approach, 1's are represents for (1,1), (1,3), (1,5)... and (1,2),(1,4),(1,6)..., and 2's are represents for (2,2),(2,4),(2,6)... and (2,1),(2,3),(2,5)... which all are explains the two-subfield consideration.

1	2	1	2	1	2	1	2	1	2	1	2	1	2
2	1	2	1	2	1	2	1	2	1	2	1	2	1
1	2	1	2	1	2	1	2	1	2	1	2	1	2
2	1	2	1	2	1	2	1	2	1	2	1	2	1
1	2	1	2	1	2	1	2	1	2	1	2	1	2
2	1	2	1	2	1	2	1	2	1	2	1	2	1
1	2	1	2	1	2	1	2	1	2	1	2	1	2

To create the algorithm, let's assume that the 8-neighbors around each 3x3 fields' main pixel go in order. Depending on the optionality, 2 or 3 functions should be implemented.

The first function is B(p) function which is counts the numbers of 1’s around the pixel. C(p) function calculates the number of connected 1's around the pixel which is in the center. Also, to calculate C(p), calculating T(p) would be helpful. T(p) function calculates connected 1's around the pixel excluding the corners. If the number of corners that change from 0 to 1 is set to 1, then the function T(p) calculates the value of C(p).

Each subfield requires two conditions which are deletion and augmentation. If the centered pixel equals to 1 deletion condition will occur if the other conditions will satisfied. If the center is p equals to 0, this time augmentation condition will take place when it's conditions are satisfied.

When all these processes are finished, the algorithm shrinks the pattern from the corners and allows the objects to be counted with faster iterations. [4]

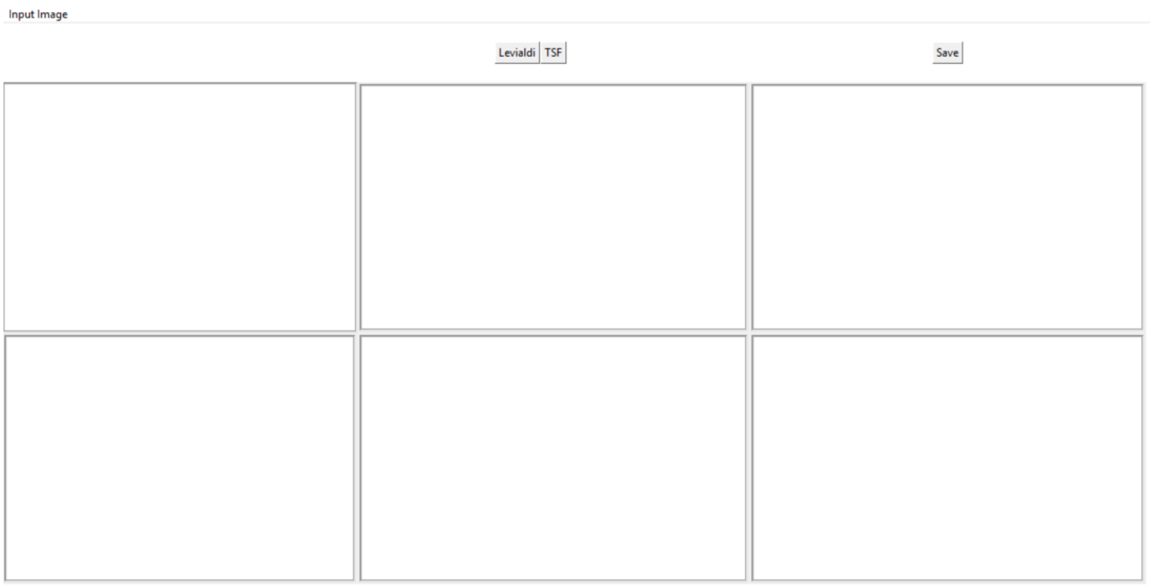
1	1	1
1	1	1
1	1	1

0.	1	0.
1	1.	1
0.	1	0.

0	0.	0
0.	1	0.
0	0.	0

Work:

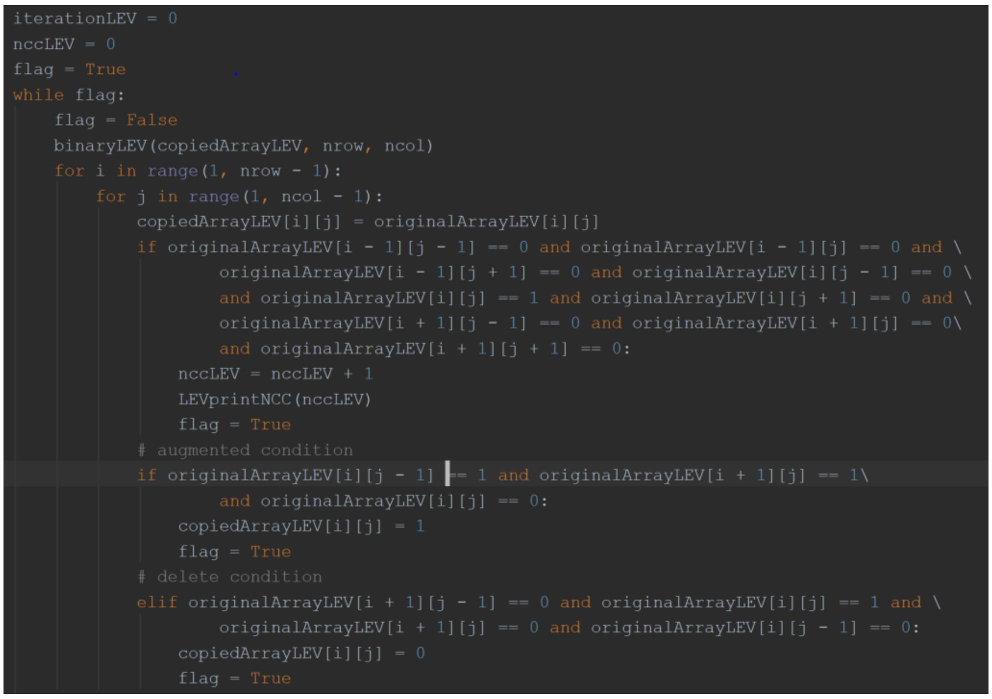
In order to implement the GUI as below, I have organized and added certain functions in the sample GUI. I implemented a menu for Drop-Down Menu, Canvas for processing in white boxes such as showing Levialdi and TSF algorithms' applications on the binary image, Button to call functions with buttons, Label for insert buttons and canvases and open the image in the GUI. Also, for creating any GUI, writing must starts root(or any other parameter name).Tk() and ends with .mainloop() to be able to continue.



In Levialdi’s Algorithm, first the boolean flag provides the continues of the while-loop. If there are not any interations exists, it’ll stop the loop. The for-loop in while-loop allows us to navigate the whole array. We also framed the outside of the image to find the objects in the corners, and we put the changes we made into the array we framed.

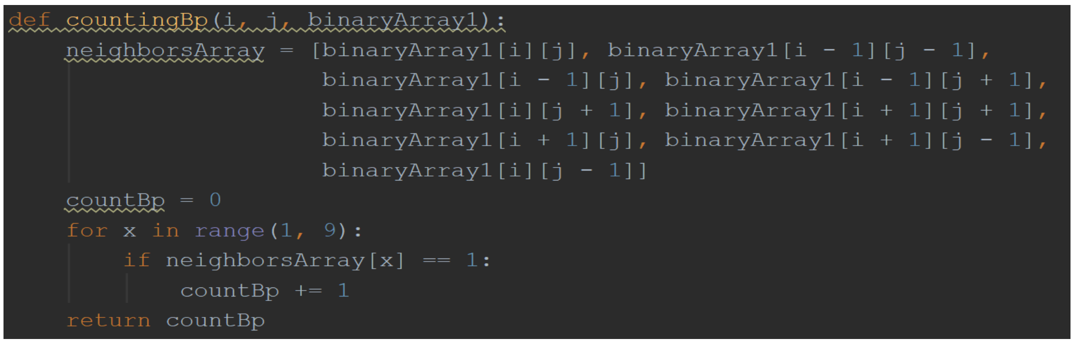
In the first if condition, it checks whether the point is isolated or not. If it's isolated, it erases the point and increases the object number.

Lastly, second and third conditions check for deletion and augmentation conditions if it provides by the point, it will occur. If it's not, that means no iteration left, while-loop will end.



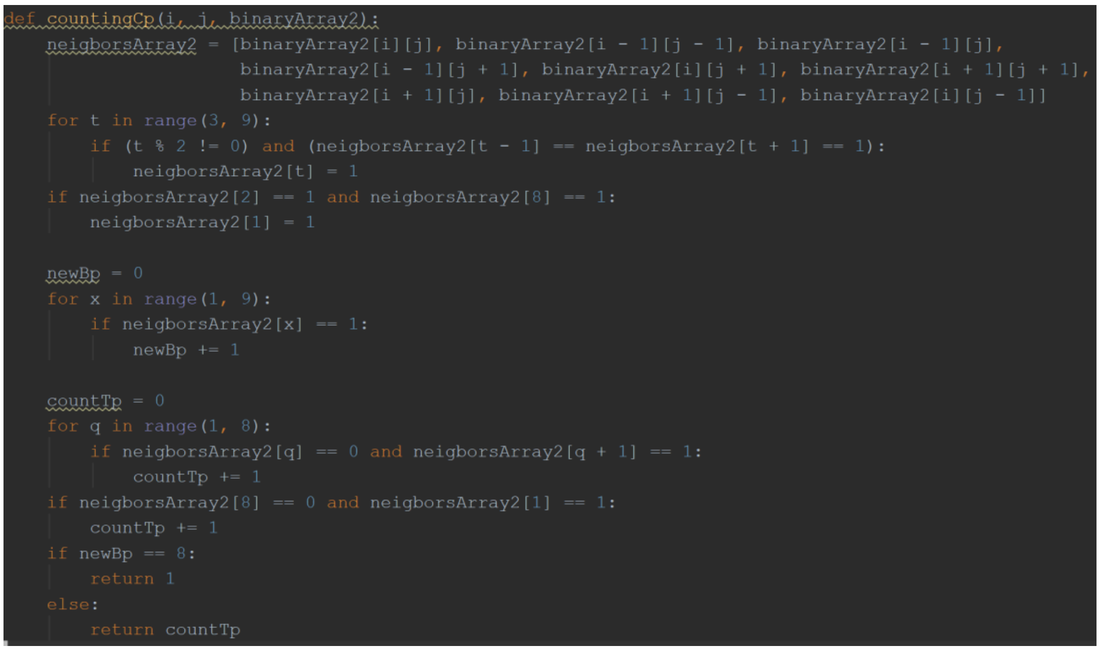
```python
iterationLEV = 0
nccLEV = 0
flag = True
while flag:
    flag = False
    binaryLEV(copiedArrayLEV, nrow, ncol)
    for i in range(1, nrow - 1):
        for j in range(1, ncol - 1):
            copiedArrayLEV[i][j] = originalArrayLEV[i][j]
            if originalArrayLEV[i - 1][j - 1] == 0 and originalArrayLEV[i - 1][j] == 0 and \
                    originalArrayLEV[i - 1][j + 1] == 0 and originalArrayLEV[i][j - 1] == 0 \
                    and originalArrayLEV[i][j] == 1 and originalArrayLEV[i][j + 1] == 0 and \
                    originalArrayLEV[i + 1][j - 1] == 0 and originalArrayLEV[i + 1][j] == 0\
                    and originalArrayLEV[i + 1][j + 1] == 0:
                nccLEV = nccLEV + 1
                LEVprintNCC(nccLEV)
                flag = True
            # augmented condition
            if originalArrayLEV[i][j - 1] == 1 and originalArrayLEV[i + 1][j] == 1\
                    and originalArrayLEV[i][j] == 0:
                copiedArrayLEV[i][j] = 1
                flag = True
            # delete condition
            elif originalArrayLEV[i + 1][j - 1] == 0 and originalArrayLEV[i][j] == 1 and \
                    originalArrayLEV[i + 1][j] == 0 and originalArrayLEV[i][j - 1] == 0:
                copiedArrayLEV[i][j] = 0
                flag = True
```

In TSF Algorithm, all functions create new array for making their calculations understandable. At the same time, in order to count the objects in the corners, binary image framed by the new array. All of the operations occur on it.



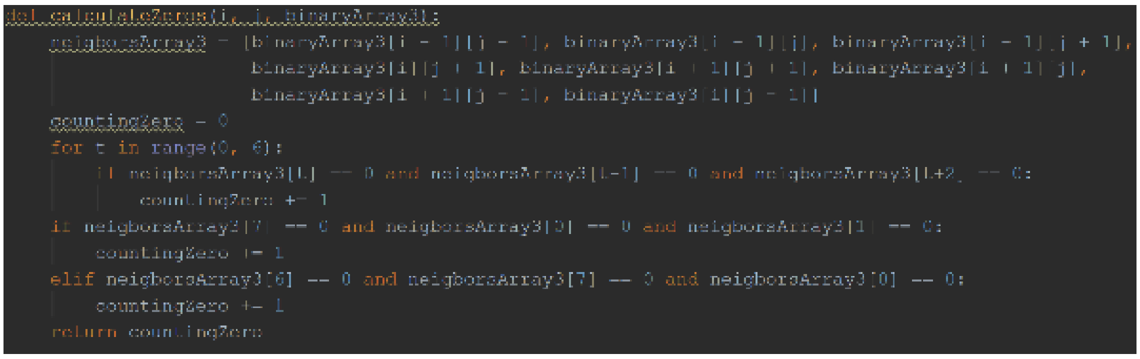
```python
def countingBp(i, j, binaryArray1):
    neighborsArray = [binaryArray1[i][j], binaryArray1[i - 1][j - 1],
                      binaryArray1[i - 1][j], binaryArray1[i - 1][j + 1],
                      binaryArray1[i][j + 1], binaryArray1[i + 1][j + 1],
                      binaryArray1[i + 1][j], binaryArray1[i + 1][j - 1],
                      binaryArray1[i][j - 1]]
    countBp = 0
    for x in range(1, 9):
        if neighborsArray[x] == 1:
            countBp += 1
    return countBp
```

In order to calculate C(p), first, calculating the T(p) is a more efficient way. While calculating the T(p), if there are any 0 to 1 changes occurs on the corners, that corner will be 1 and T(p) will be equal to C(p).



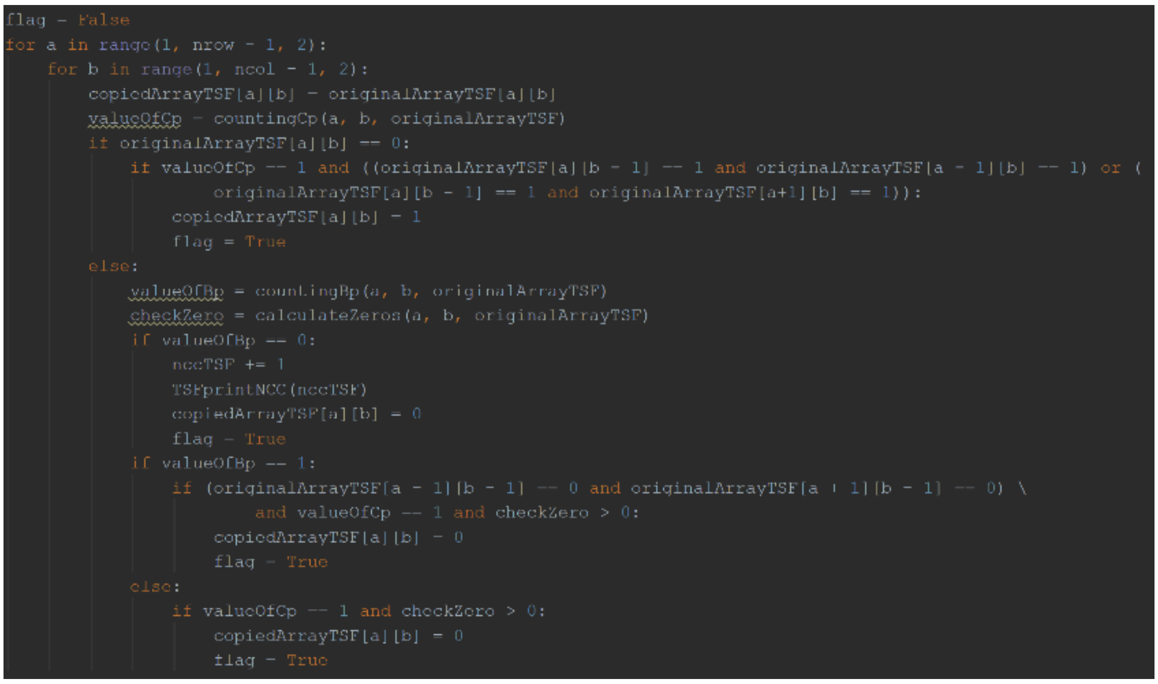
```python
def countingCp(i, j, binaryArray2):
    neigborsArray2 = [binaryArray2[i][j], binaryArray2[i - 1][j - 1], binaryArray2[i - 1][j],
                      binaryArray2[i - 1][j + 1], binaryArray2[i][j + 1], binaryArray2[i + 1][j + 1],
                      binaryArray2[i + 1][j], binaryArray2[i + 1][j - 1], binaryArray2[i][j - 1]]
    for t in range(3, 9):
        if (t % 2 != 0) and (neigborsArray2[t - 1] == neigborsArray2[t + 1] == 1):
            neigborsArray2[t] = 1
    if neigborsArray2[2] == 1 and neigborsArray2[8] == 1:
        neigborsArray2[1] = 1

    newBp = 0
    for x in range(1, 9):
        if neigborsArray2[x] == 1:
            newBp += 1

    countTp = 0
    for q in range(1, 8):
        if neigborsArray2[q] == 0 and neigborsArray2[q + 1] == 1:
            countTp += 1
    if neigborsArray2[8] == 0 and neigborsArray2[1] == 1:
        countTp += 1
    if newBp == 8:
        return 1
    else:
        return countTp
```

Calculating zero function will calculate is there any 3 or greater than 3 connected zeros in the 8-neighbors, if there is it will be returning a value.



```python
def calculateZeros(i, j, binaryArray3):
    neigborsArray3 = [binaryArray3[i - 1][j - 1], binaryArray3[i - 1][j], binaryArray3[i - 1][j + 1],
                      binaryArray3[i][j + 1], binaryArray3[i + 1][j + 1], binaryArray3[i + 1][j],
                      binaryArray3[i + 1][j - 1], binaryArray3[i][j - 1]]
    countingZero = 0
    for t in range(0, 6):
        if neigborsArray3[t] == 0 and neigborsArray3[t+1] == 0 and neigborsArray3[t+2] == 0:
            countingZero += 1
    if neigborsArray3[7] == 0 and neigborsArray3[0] == 0 and neigborsArray3[1] == 0:
        countingZero += 1
    elif neigborsArray3[6] == 0 and neigborsArray3[7] == 0 and neigborsArray3[0] == 0:
        countingZero += 1
    return countingZero
```

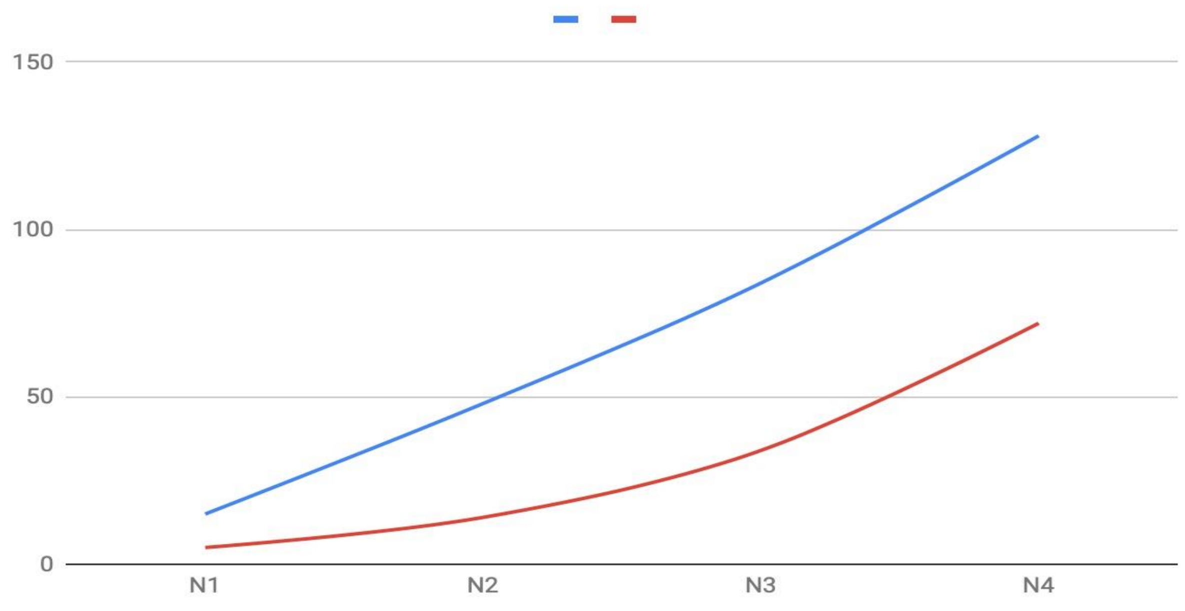
This is the half of one of the subfields, the other ones contain the same. If deletion conditions are satisfied, the center point will be one to zero, if augmentation conditions are satisfied, the center point will be zero to one. Otherwise, boolean returns False and while-loop will end because there are no iterations left.

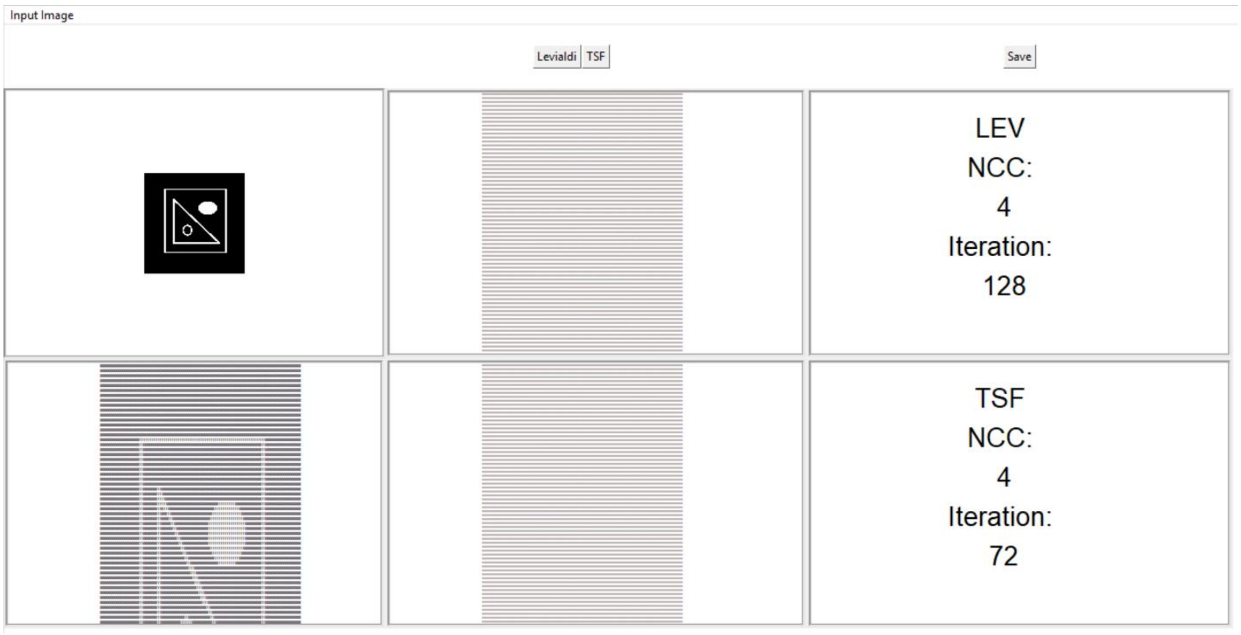


```python
flag = False
for a in range(1, nrow - 1, 2):
    for b in range(1, ncol - 1, 2):
        copiedArrayTSF[a][b] = originalArrayTSF[a][b]
        valueOfCp = countingCp(a, b, originalArrayTSF)
        if originalArrayTSF[a][b] == 0:
            if valueOfCp == 1 and ((originalArrayTSF[a][b - 1] == 1 and originalArrayTSF[a - 1][b] == 1) or (
                    originalArrayTSF[a][b - 1] == 1 and originalArrayTSF[a+1][b] == 1)):
                copiedArrayTSF[a][b] = 1
                flag = True
        else:
            valueOfBp = countingBp(a, b, originalArrayTSF)
            checkZero = calculateZeros(a, b, originalArrayTSF)
            if valueOfBp == 0:
                nccTSF += 1
                TSFprintNCC(nccTSF)
                copiedArrayTSF[a][b] = 0
                flag = True
            if valueOfBp == 1:
                if (originalArrayTSF[a - 1][b - 1] == 0 and originalArrayTSF[a + 1][b - 1] == 0) \
                        and valueOfCp == 1 and checkZero > 0:
                    copiedArrayTSF[a][b] = 0
                    flag = True
            else:
                if valueOfCp == 1 and checkZero > 0:
                    copiedArrayTSF[a][b] = 0
                    flag = True
```

Results:

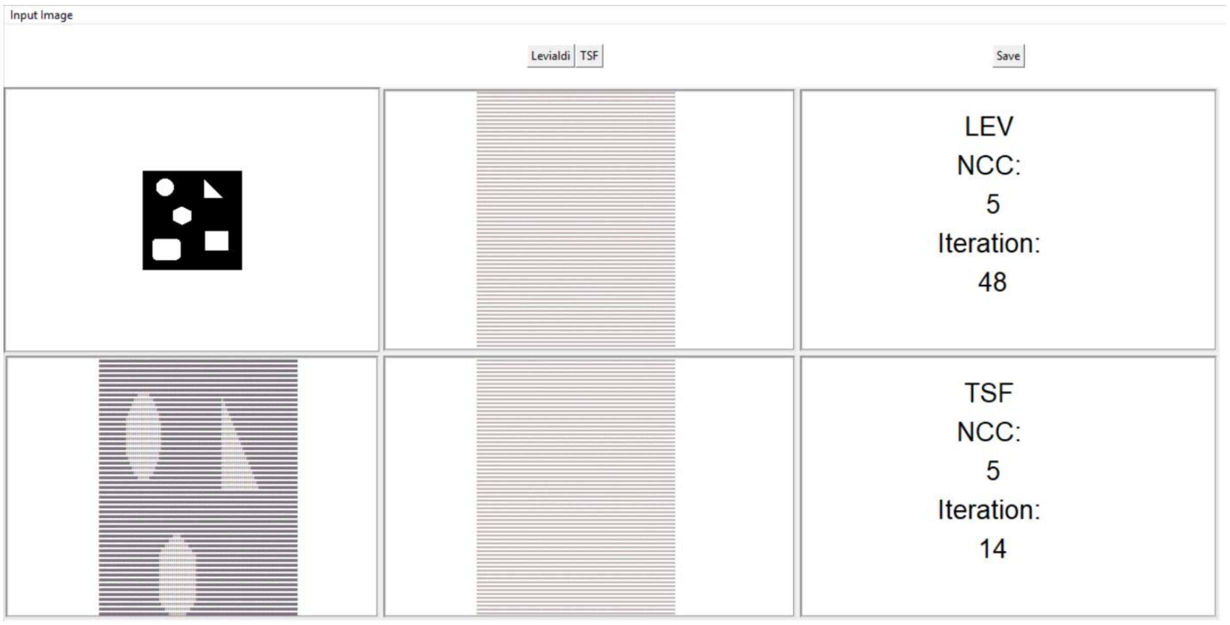
As the number of objects and image size increases, the iterations of Levialdi are slowed down, while the TSF maintains the number of iterations in small images. Small iteration number means that process happens fastly. In results, TSF Algorithm is more efficient than Levialdi's Algorithm because it's speed is much faster than Levialdi's. This will provide efficiency and gaining time while working on complex and big size images.





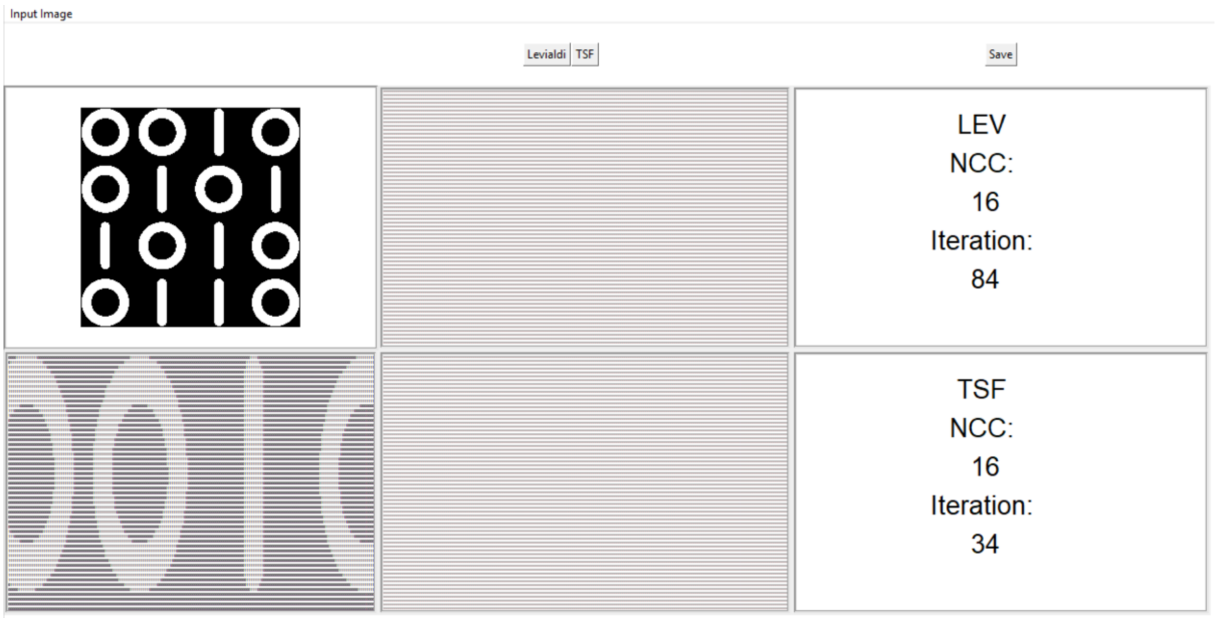
LEV
NCC:
4
Iteration:
128

TSF
NCC:
4
Iteration:
72



LEV
NCC:
5
Iteration:
48

TSF
NCC:
5
Iteration:
14



LEV
NCC:
16
Iteration:
84

TSF
NCC:
16
Iteration:
34

Contribution:

During working on this project, I've learned Image Processing, GUI, what Parallel Algorithm is, and how to count the objects in the given image. Learning different algorithms has improved my perspective. I also learned how to efficiently solve the errors which I received in the code.

Conclusion:

In conclusion, The TSF algorithm is provided with much less iteration as it is shrinking all the corners instead of being shrunk towards the upper right corner like the Levialdi's algorithm. The fewer iteration results in faster processing. Another reason for the rapidity of the iterations in TSF's algorithm is that it uses parallel algorithm while reducing the corners at the same time.

In addition, it was observed that Levialdi was working more slowly as the size of the images grew and the number of objects in it increased. For large and complex images, the TSF algorithm's fast working will provide more efficiency and usability.

References:

[1] A. P and J. Rangole, "Literature Review on Object Counting Using Image Processing Techniques", *Rroij.com*. [Online]. Available: http://www.rroij.com/open-access/literature-review-on-object-counting-usingimage-processing-techniques.php?aid=42706. [Accessed: 05- Mar- 2019].

[2] V. Voevodin and A. Antonov, *What do we need to know about parallel algorithms and their efficient implementation?*, 1st ed. https://grid.cs.gsu.edu/~tcpp/curriculum/?q=system/files/Ch02_1.pdf, p. 36. [Accessed: 05- Mar- 2019].

[3] S. Levialdi, “On Shrinking Binary Picture Pattern”. Available: http://delivery.acm.org/10.1145/370000/361240/p7-levialdi.pdf?ip=213.14.214.236&id=361240&acc=ACTIVE%20SERVICE&key=956257EA1AE1732

3%2E2588E645DDE70CC1%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&__acm__=1551799006_4e90f36a46149a4c2e0c854edfe7634b. [Accessed: 05- Mar- 2019].

[4] M. Gokmen and R. W. Hall, “Parallel Shrinking Algorithms Using 2-Subfields Approaches” Available: https://mef.blackboard.com/bbcswebdav/pid-200227-dt-content-rid-6864715_1/courses/2018-2019-02-COMP204-01/Parallel%20shrinking%20algorithms%20using%202-subfields%20approaches%20%281%29.pdf. [Accessed: 05- Mar- 2019].