Progetto di Ingegneria Informatica

Implementation of a Region Growing Algorithm for Burned Area Mapping from Sentinel-2 Data

Thomas Martinoli
Matricola 952907

Tutor: ***prof.ssa Giovanna Venuti***

Accademic year 2021/2022

TABLE OF CONTENTS

1. Overview

The following chapter contains the main information on the work carried out: initially the objective is reported, then all the specifications relating to the machine and the software used are provided, finally the study case considered is introduced in detail.

1.1 Target

The purpose of the work is the development of an algorithm for Region Growing (RG), that will be introduced in the already existing computation process of Burned Area (BA). The estimation of BA and the creation of Burned Map (BM) is well described in the article "*A Burned Area Mapping Algorithm for Sentinel-2 Data Based on Approximate Reasoning and Region Growing"[1],* and it is the starting point for this research.

In the following paragraphs it will be described step by step the approach and the idea that has guided the implementation of the algorithm. Result will be presented and commented together with a test on a real scenario with Sentinel-2 images for fire events occurred in Portugal in the year 2017.

1.2 Machine specification

The machine that is used to develop the work and run the process has the following characteristics:

- Processor: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz - 2.81 GHz;
- RAM: 16,0 GB;
- Operative system 64 bit.

1.3 Software and libraries specification

The algorithm is implemented using Python (*version 3.9.13*) in Anaconda environment (*version 2.3.1*). This choice is due to the fact that it is a high-level, general-purpose and open-source programming language. Furthermore, it leaves the possibility in the future to build a plug-in in QGIS. QGIS (*version 3.26*) it also used in this work as detailed below.

In order to develop the code, the following Python's libraries are involved:

- Numpy (*version 1.23.4*), it is a package for scientific computing in Python. It provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more. [2]
- Time, it is a built-in module that provides various functions to manipulate time values
- Matplotlib (*version 3.6.2*), it is a comprehensive library for creating static, animated, and interactive visualizations in Python. [3]
- Rasterio (version 1.3.3), it is a library that reads and writes raster (in GeoTIFF and other formats) dataset that are used in GIS (Geographic Information Systems). [4]
- Os, this built-in module provides a portable way of using operating system dependent functionality. [5]
- Copy, it is a built-in module that offers shallow and deep copying operations.

1.4 Repository

All codes that were developed and the instruction in order to set properly the Anaconda environment are in a repository in GitHub at the following link:
https://github.com/ThomasMartinoli/Project-GeoInf.git

1.5 Introduction

As already mentioned, the starting point is the article describing the algorithm developed at IREA CNR to map Burned Area (BA) from Sentinel-2 images and based on the theory of fuzzy sets and convergence of evidence *[1]*. So, a brief description of the main steps that lead the generation of the Burned Map (BM) is given for completeness.
The principal phases are:

a) Input Selection:

two *Sentinel-2*'s images of the area of interest, at different time, are collected. For each images significant bands are selected and differences between them are computed. This crucial phase is at the base of the entire process: the selection of appropriate inputs makes possible the classification of the two images, in order to identify the areas affected by fire at different periods.

b) Membership degree computation:

applying the Fuzy theory on the results of the previous step, it is possible to define a Membership Functions (MF) for each pixel, that for each pixel quantifies the degree of membership (MD) to the burned area category. The MD is a continuous value between *0* and *1* that represent how much a pixel belong to a specific class: *0* means that the pixel is more likely to be Unburned (U), *1*, instead, Burned (B).

c) Ordered Weight Avarege (OWA) computation:

due to the fact that the final result should be a binary map in which each pixel contains only one information: Unburned or Burned. Indeed, up to step 2, it is computed a value of MD for each pixel and input layer (n); so it is necessary to convert a multi-dimensional information to a single layer (a sort of synthetic information on the likelihood of begin burned for each pixel), that is the global evidence of burn. In order to do this, OWA operators are used to integrate input information derived from multiple bands. $OWAs$ belong to a parameterized family of soft-mean-like aggregation operators. Different operators were used to represent attitudes ranging between:

- pessimistic (OWA_{or} considers the maximum extent of the phenomenon to minimize the chance of underestimating: modeling a compensative aggregation to integrate complementarities of multiple criteria);
- optimistic (OWA_{and} considers the minimum extent of the phenomenon to minimize the chance of overestimating: modeling a concurrent aggregation to integrate mutual reinforcement of multiple criteria).

Layers of global evidence derived with different $OWAs$ will be input to a region growing.

d) Region Growing:

This last phase is based on two different layers: *Seed* and *Grow*. The *Seed layer* is generated computing the OWA_{and}, and it is fundamental because allows to reduce the commission error; instead, the *grow layer* is obtained starting from the OWA_{or}, and it allows to reduce the omission error. In this way much accuracy is obtained in the final result. As the name suggest the basic idea is to make grow the *grow layer* on the seeds (that belong to the *seed layer)*. How to do this and much more details are introduced later.

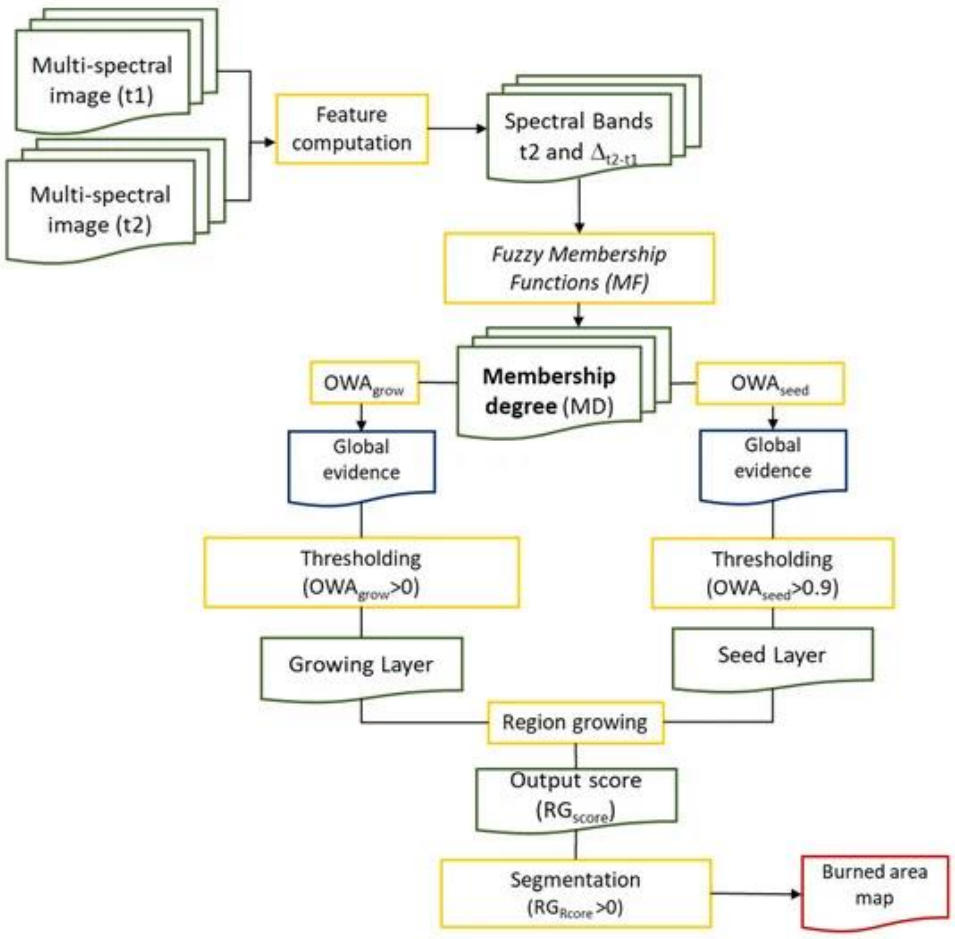


Figure 1. Flowchart of the processing steps from input S2 MSI images to generate BA maps.

2. Implementation

This paragraph highlights the last point of the previous section and tries to answer to those questions. As already said, the scope is: to make the *grow layer* grow starting from seed pixels extracted from the *seed layer*.

So, the first step is the computation of the two layers: *seed layer* and *grow layer*. The first one is obtained after the discretization of the OWA_{and}: if the value inside the pixel is greater, or equal, to *0.9*, it is assigned *1*, otherwise *0*. The same procedure is applied to the OWA_{or} in order to obtain the *grow layer*, in this case the threshold considered is *0.1*. After this preliminary step, that it is done directly in QGIS with the raster calculator function, the images (in *.tif* format) are imported in Python and it will be considered as a binary matrix, in this way an element in the matrix correspond to an image's pixel.

The next step in the algorithm is to select one seed at time (where the value in the *seed layer* is equal to *1*) and look to the eight neighbors. Once they are selected, the corresponding value inside the *grow layer* is considered, if it is *1*, that neighbor pixel becomes a new seed. This procedure must be repeated until no new seeds are found or the limit of the map is reached.

Two computation approaches were tested to implement the code:

- scrolling the matrix Pixel by Pixel;
- summing the layers.

2.1 Structure of data

Python offers the possibility to create multidimensional array (to be more precise it will be used a multidimensional matrix) thanks to the *numpy* library.

So, it is possible to well organize our data in the following way:

- the first matrix contains the seed layer with all bands related to the Raster (in our case *1*);
- the second one the grow layer with all bands related to the Raster (in our case *1*);
- the last one an index term, that will be useful during the implementation of the code.

The structure is the following one:

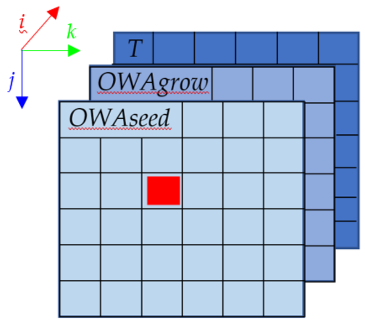


Figure 2. Data's structure

Therefore, with the term *i* it is possible to select one of the three matrices (notice that Python start counting from *0*). Instead, with *j* and *k* is selected a cell (row and column) of the matrix.

For example, looking at the previous image (*Figure 2*), the red cell can be access using the following sequence: [*i*][*band,j,k*] = [*0*][*0,2,2*].

2.2 Scrolling the matrix Pixel by Pixel

Scrolling through the matrix, that represent the seed layer, thank to two '*for*' cycles (appendix A code's *line 15-16*), the seeds (elements with value *1*) are identified (aA *l.20*).

At this point, when a seed is found, its eight neighbors are taken in account (aA *l. 26-81*).

Some conditions must be considered in order to avoid errors during the execution of the code:

- an element in the first column has not neighbors on the left;
- an element in the last column has not neighbors on the right;
- an element in the first row has not neighbors above;
- an element in the last row has not neighbor below;

For the selected elements, if it is not a seed (*OWAseed = 0*), the values inside the grow matrix are considered (aA *l. 26-81*).

If the condition is verified (*OWAgrow = 1*) the element is added to the seed matrix, and before proceeding its state is modified (inside each '*if*' condition).

Thanks to this shrewdness it is possible to know at which step a new seed is inserted: it is sufficient to modify the value *-1* with a counter that indicates the number of cycles, and so when a value in *T* different from *-1* is read, that element must not be considered again.

Losing this information, when a new seed is inserted, and considering only the constrain if a pixel is a seed or not (new or old), it is possible to improve the computational time of the entire process. This happens because it is not necessary to read and write/overwrite in memory the matrix *T* (this improvement is shown in a graph in paragraph *3*).

When each cell of the seed matrix has been considered, the entire procedure is repeated with the new seed matrix, this is done until no new seeds are added between two consecutive cycle (aA *l. 11*).

2.3 Summing the layers

Here, the approach is more intuitive and direct with respect to the previous case.

The idea is to sum the *seed layer* with the *grow layer*, in order to generate a new *seed layer*.

Preliminary to this operation, it is necessary to prepare the *neighbors matrix*: the seeds are selected from the *seed matrix* and the eight neighbors of each seed are set to *1* (thanks to the function: *CercaVicini*).

From the article [1] an assumption may be derived: if a pixel satisfies the condition OWA_{and} = *1*, then for sure it will be satisfying also the following condition OWA_{or} = *1*. This means that element will be equal to *1* in both layers.

This consideration becomes fundamental for this method, but it is better to highlight the fact that it may be considered with the approach proposed by the article [1] and it is not a generic hypothesis for RG algorithms, so in any other cases must be verified.

So, there are two values (*0* and *1*) that describe three scenarios in the input layers:

- *0* indicates that a pixel is unburned in both layers (green);
- *1* indicates that a pixel is burned in the *grow layer* (red);
- *1* indicates that a pixel is burned (a seed), but also represent a seed's neighbor in the *seed layer* (red).

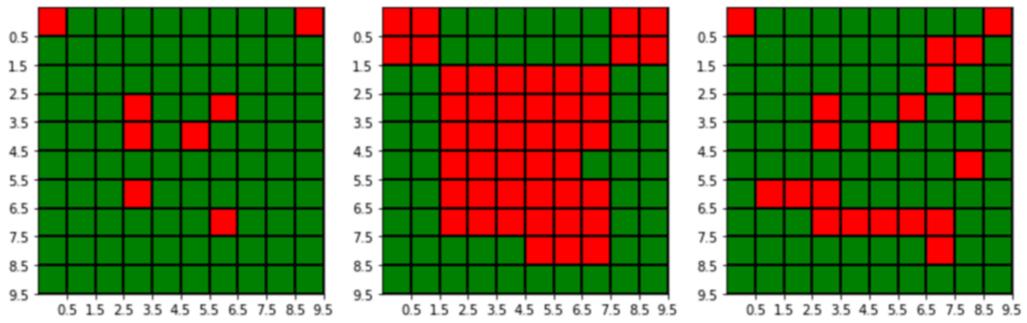


Figure 3. *In order: seed matrix, neighbor matrix, grow matrix*

Now it is possible to proceed summing the two matrixes (aA *l. 16*). The result that is obtained is a matrix with three values:

- *0* means that element is does not satisfy the condition for being a seed nor for growing (green);
- *1* means that element is burned only for the *grow layer* or that is a neighbor for a seed (red);
- *2* means that element is burned for each layer and so must be considered (blue).

This result is then transformed in binary matrix in order to generate the layer with the new seeds (aA *l. 18*).

At this point the procedure is repeated until the number of seeds added in two consecutive iterations does not change (aA *l. 11*).

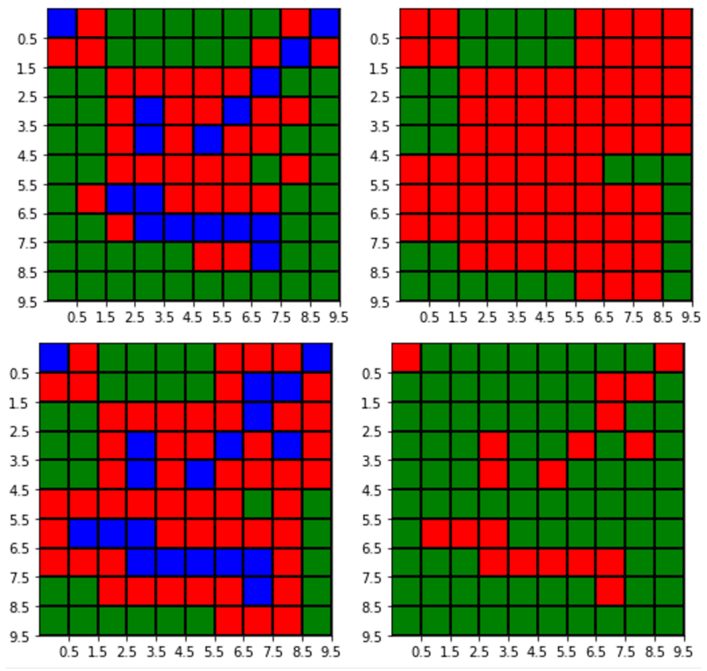


Figure 4. *In order: sum at cycle 1, neighbors and seed at cycle 2, sum at cycle 2, final result*

3. Test and Discussion

This section shows the results of the two approaches implemented in Phyton. In particular, it was analyzed some performance indicators to highlight whether one of the two approaches could be preferred especially for processing large rasters.

The process time are quite similar for small images, but the first method (*scrolling the matrix Pixel by Pixel*) offers a faster solution when the image's size starts to increase.

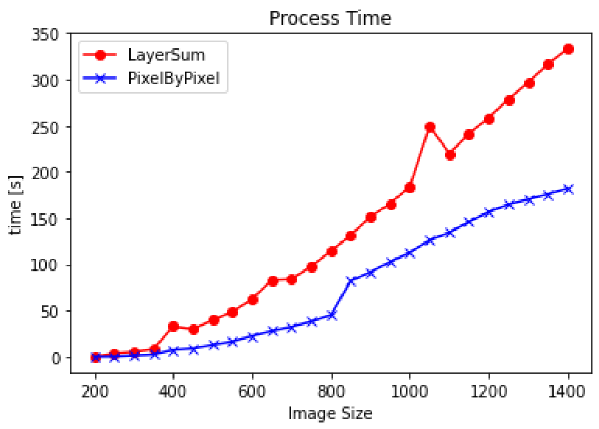


Figure 5. Process time for Images with size that goes from 200x200 to 1400x1400 pixel (step 50)

As already anticipated in section *2.2*, the computational time can be reduced further, as shown in *Figure 6*. In order to do this the matrix *T* is not considered, and only the following condition is used: if a neighbor is a seed, the process goes ahead and does not consider it. This information is red in the *seed matrix*, that is updated at each step of each cycle, as already explained in section *2.2*.

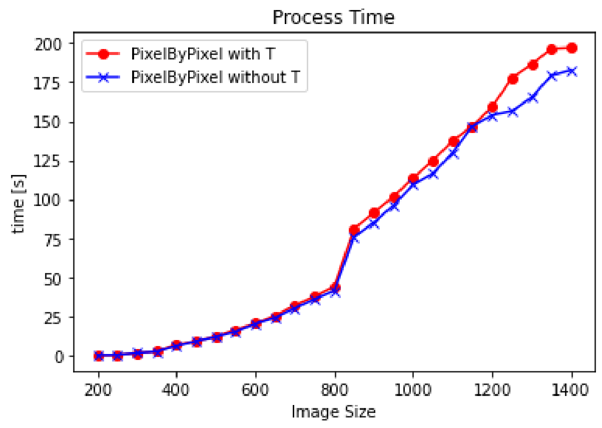


Figure 6. Process time using the first method with T and without on images with size that goes from 200x200 to 1400x1400 pixel (step 50)

As a second test I analyzed the number of seeds for each iteration and *Figure 7* shows the result for the two methods. Both approaches scroll the matrix at each cycle, however the first method adds immediately a new seed. If this new element is on the right, or it is below with respect to the analyzed seed, it will be immediately considered during the scrolling of the matrix, and not on the next scrolling cycle.

This aspect is much more evident looking at the convergence of the two methods. In fact, after one cycle the number of seeds is higher in the first approach and is much nearer to the final output. Visually, the faster convergence is shown in *Figure 8* for the raster layer obtained by the two approaches at the first iteration.

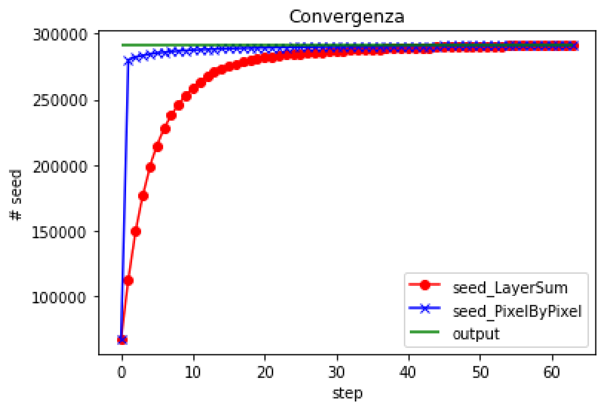


Figure 8. Number of seeds at each scrolling cycle of the seed matrix (image's size 700x700 pixel)

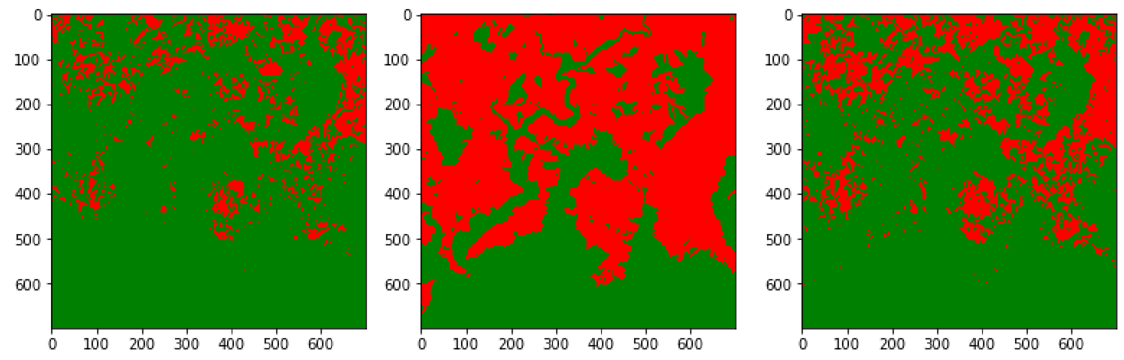
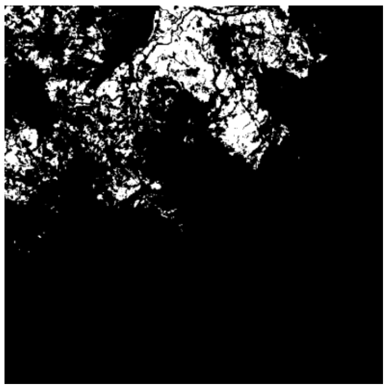
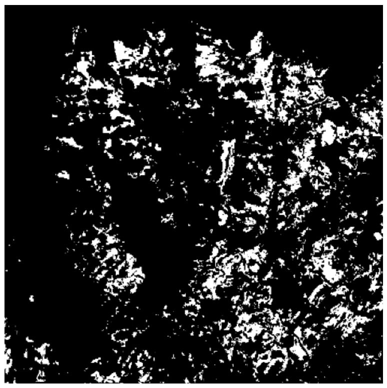


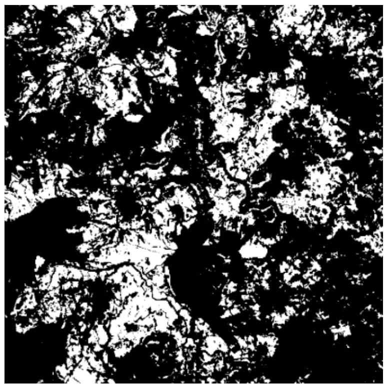
Figure 7. In order: seed layer, seed layer after one cycle with first method, seed layer after one cycle with the second method

I also tested if there is relationship with respect to the initial number of seeds and the process time. In order to do this the two approaches are applied to five random regions of 700x700 pixel.

clip 1


clip 2


clip 3


clip 4

clip 5

Figure 9. Clips of 700x700 pixel of seed layer. In white Burned pixel, in black Unburned ones

From the trend of the graph (*Figure 10*) it is not possible to conclude that there is a dependence between the two quantities, but the time will depend also on how the seeds are distributed in the considered area. For example, looking at clip 2, it has a high concentration of seeds in the upper left area and needs less time to analyze it.

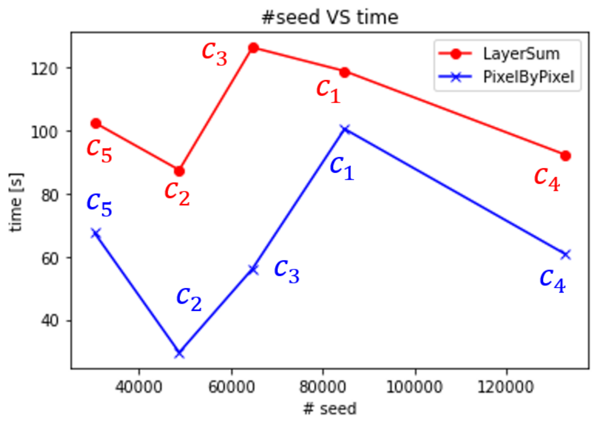


Figure 10. Process time with respect to each clip (c_i)

Finally, the result that are obtained with both methods are compared with the expected output (considered as reference), that was derived by applied the region growing algorithm in a different software (ENVI Exelis Visual Information Solution, Boulder, Colorado). Due to the fact that the two approaches give the same result for simplicity is reported one general case.

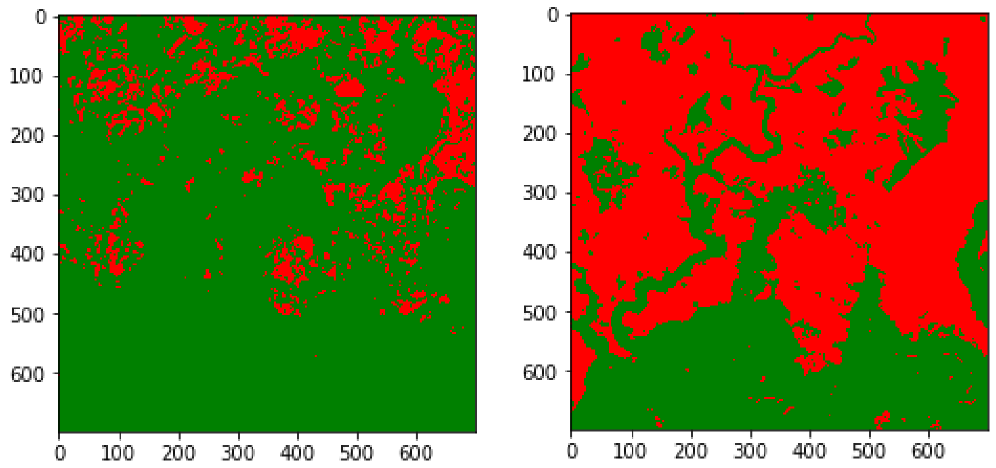


Figure 11. In order: seed layer, grow layer. In green Unburned area (700x700 pixel), in red Burned

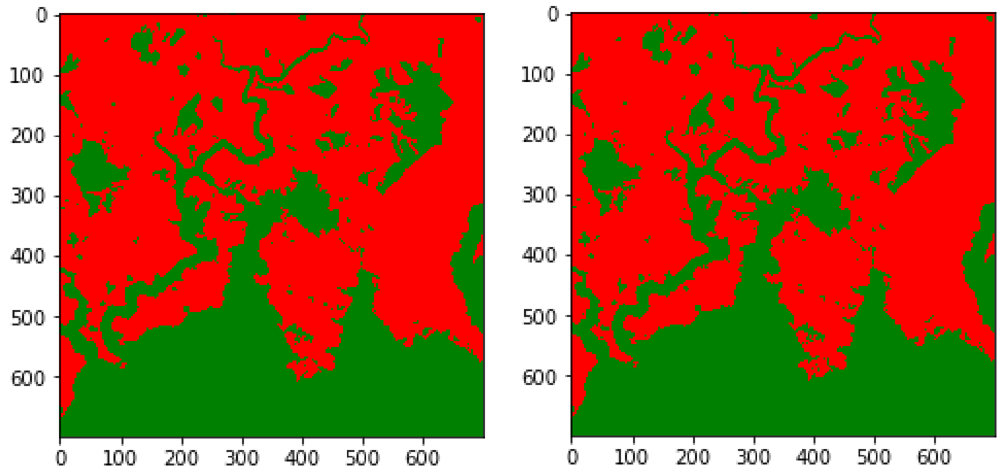


Figure 12. In order: Official Result, Result with the two approaches

To conclude the confusion matrix is computed:

		REFERENCE	
		Burned	Unburned
ALGORITHM	Burned	*289467*	*1647*
	Unburned	*2664*	*196222*

From that is possible to compute the following errors:

- commission: *0.57%*
- omission: *0.91%*

4. Conclusion

Although the sum layer method is more intuitive and immediately understandable, it turns out to be slower, this because in the *PixelByPixel* method one does not wait for the end of an entire cycle (reading the entire matrix) to update the *seed matrix,* but immediately updates the *seed matrix* , in this way the new information is available in the next step.

The *PixelByPixel* method can be further optimized by neglecting the *T* matrix.

Furthermore, the *PixelByPixel* method appears to reach convergence (final number of seeds) more rapidly, so in a possible preliminary analysis the method could provide realistic and meaningful information after a few cycles. This would allow to analyse areas of greater extension in less time.

References

1. Sali, M.; Piaser, E.; Boschetti, M.; Brivio, P.A.; Sona, G.; Bordogna, G.; Stroppiana, D. A Burned Area Mapping Algorithm for Sentinel-2 Data Based on Approximate Reasoning and Region Growing. *Remote Sens*. **2021**, 13, 2214. [Link]
2. Documentation numpy [Link]
3. Documentation matplotlib [Link]
4. Documentation rasterio [Link]
5. Documentation os [Link]

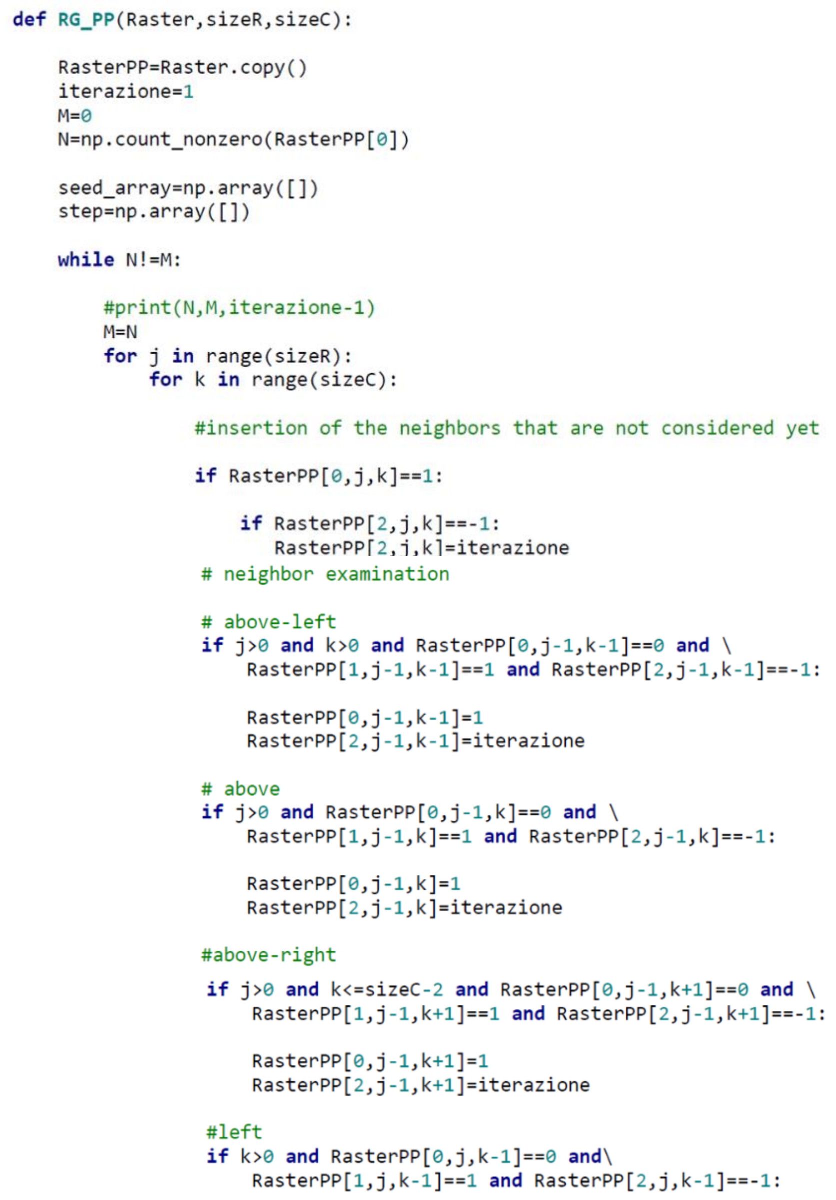
Appendix A

Inside this section are reported the principal part of the codes that could be help the reader to understand.

I. Pixelbypixel

Legend

- M: is set to 0 (*l. 5*). It is a variable used to memorize the number of seed before the research (*l. 14*), and then it is compared with the new seed number N.
- N: is the number of seed that are in the *seed matrix* (elements different from zero).
- Iterazione: is the number of cycles.



```
1   def RG_PP(Raster,sizeR,sizeC):
2
3       RasterPP=Raster.copy()
4       iterazione=1
5       M=0
6       N=np.count_nonzero(RasterPP[0])
7
8       seed_array=np.array([])
9       step=np.array([])
10
11      while N!=M:
12
13          #print(N,M,iterazione-1)
14          M=N
15          for j in range(sizeR):
16              for k in range(sizeC):
17
18                  #insertion of the neighbors that are not considered yet
19
20                  if RasterPP[0,j,k]==1:
21
22                      if RasterPP[2,j,k]==-1:
23                         RasterPP[2,j,k]=iterazione
24                  # neighbor examination
25
26                  # above-left
27                  if j>0 and k>0 and RasterPP[0,j-1,k-1]==0 and \
28                      RasterPP[1,j-1,k-1]==1 and RasterPP[2,j-1,k-1]==-1:
29
30                      RasterPP[0,j-1,k-1]=1
31                      RasterPP[2,j-1,k-1]=iterazione
32
33                  # above
34                  if j>0 and RasterPP[0,j-1,k]==0 and \
35                      RasterPP[1,j-1,k]==1 and RasterPP[2,j-1,k]==-1:
36
37                      RasterPP[0,j-1,k]=1
38                      RasterPP[2,j-1,k]=iterazione
39
40
41                  #above-right
42                  if j>0 and k<=sizeC-2 and RasterPP[0,j-1,k+1]==0 and \
43                      RasterPP[1,j-1,k+1]==1 and RasterPP[2,j-1,k+1]==-1:
44
45                      RasterPP[0,j-1,k+1]=1
46                      RasterPP[2,j-1,k+1]=iterazione
47
48                  #left
49                  if k>0 and RasterPP[0,j,k-1]==0 and\
50                      RasterPP[1,j,k-1]==1 and RasterPP[2,j,k-1]==-1:
51
```

```
                        RasterPP[0,j,k-1]=1
                        RasterPP[2,j,k-1]=iterazione

                    #right
                    if k<=sizeC-2 and RasterPP[0,j,k+1]==0 and \
                        RasterPP[1,j,k+1]==1 and RasterPP[2,j,k+1]==-1:

                        RasterPP[0,j,k+1]=1
                        RasterPP[2,j,k+1]=iterazione

                    #below-left
                    if j<=sizeR-2 and k>0 and RasterPP[0,j+1,k-1]==0 and \
                        RasterPP[1,j+1,k-1]==1 and RasterPP[2,j+1,k-1]==-1:

                        RasterPP[0,j+1,k-1]=1
                        RasterPP[2,j+1,k-1]=iterazione

                    #below
                    if j<=sizeR-2 and RasterPP[0,j+1,k]==0 and \
                        RasterPP[1,j+1,k]==1 and RasterPP[2,j+1,k]==-1:

                        RasterPP[0,j+1,k]=1
                        RasterPP[2,j+1,k]=iterazione

                    #below-right
                    if j<=sizeR-2 and k<=sizeC-2 and RasterPP[0,j+1,k+1]==0 \
                        and RasterPP[1,j+1,k+1]==1 and RasterPP[2,j+1,k+1]==-1:

                        RasterPP[0,j+1,k+1]=1
                        RasterPP[2,j+1,k+1]=iterazione

        step=np.append(step,iterazione-1)
        iterazione=iterazione+1

        #condition that blocks the while
        seed_array=np.append(seed_array,N)
        N=np.count_nonzero(RasterPP[0])

    return RasterPP, seed_array, step
```

II. Layer Sum

Legend

- M: is set to 0 (*l. 8*). It is a variable used to memorize the number of seed before the research (*l. 13*), and then it is compared with the new seed number N.
- N: is the number of seeds that are in the *seed matrix* (elements different from zero).
- k: is the number of cycle.

```
1   def RG_LS(Raster,sizeR,sizeC):
2
3       seed_array=np.array([])
4       iterazioni=np.array([])
5       N=np.count_nonzero(Raster[0])
6
7       Seed=Raster[0,:,:].copy()
8       M=0
9       k=0
10
11      while N!=M:
12
13          M=N
14          Vicini=CercaVicini(Seed,sizeR,sizeC)
15
16          Result=Vicini+Raster[1]
17
18          ModifiedResult= np.floor(Result/2)
19
20          Seed=ModifiedResult
21
22          seed_array=np.append(seed_array,N)
23          iterazioni=np.append(iterazioni,k)
24
25          N=np.count_nonzero(ModifiedResult)
26
27          k=k+1
28
29
30          #print(N,M,k)
31
32      return Seed, seed_array, iterazioni
```