POLITECNICO
MILANO 1863

Geoinformatics project

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

Thomas Martinoli
Matricola 952907

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

Accademic year 2021/2022

TABLE OF CONTENTS

Abstract

The purposes of the work is the implementation of an algorithm for Region Growing (RG) and the testing for its processing performance. The developed code will be later added as a module to an algorithm for Burned Area (BA) mapping from satellite multi-spectral images (Sentinel-2, *[1]*). A RG algorithm is a contextual image segmentation method that starting from initial seed points examines neighbouring pixels and determines whether these pixels should be added to the region. The process is iterative and for this reason can be computationally demanding.

The details of the BA algorithm are well described in the article "*A Burned Area Mapping Algorithm for Sentinel-2 Data Based on Approximate Reasoning and Region Growing" [2]*, 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 RG algorithm. The RG module will be tested to find the best and more efficient implementation and to quantify execution time; moreover, the module was tested on a real scenario with Sentinel-2 (S-2) images for mapping burned area from fire events occurred in Portugal in 2017.

S-2 images are freely available through the Copernicus Open Access Hub *[3]* where they are archived as 100 *km* * 100 *km* tiles; in this project, the implementation of the RG algorithm receives as input raster products derived from S-2 images (*OWA* layers, see section 1.4) that are made available by IREA CNR *[4]*.

1. Specification

1.1 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.

1.2 Software and libraries specification

The RG algorithm is implemented using Python (*version 3.9.13*) in Anaconda environment (*version 2.3.1*). Python 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 RG code, the following Python's libraries are used:

- 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; *[5]*
- 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; *[6]*
- 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); *[7]*
- Os, this built-in module provides a portable way of using operating system dependent functionality; *[8]*
- Copy, it is a built-in module that offers shallow and deep copying operations.

1.3 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

1.4 Algorithm specification

The starting point is the pixel-based 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 *[2]*. The RG module is part of the BA algorithm and it is used as a final step to reduce misclassification errors in burned area maps. So, a brief description of the main steps that lead to the generation of the Burned Area Map (BM) is given for completeness.

The principal phases are:

a) Input Feature Selection:

Two multi-spectral satellite *Sentinel-2* images of the area of interest, acquired at different time for change detection, are collected. For each image, significant bands are selected and differences between them are computed; input S2 spectral bands and their combination used as input to the classification algorithm are called *input features*. This step is at the base of the entire process: the selection of suitable inputs makes possible the classification of the two images, in order to identify the areas affected by fires at different periods. Input images should be selected with the least cloud cover in order to maximize the surface observed. Images are exploited to identify areas that burned between the two dates (also called pre- and post-fire images/dates) by relying on reflectance values and reflectance difference between the two acquisition dates.

b) Membership degree computation:

applying the Fuzzy set theory on the results of the previous step, it is possible to define a Membership Functions (MF) for each feature. MFs are applied to each pixel to compute the degree of membership (MD), that is a continuous value between *0* and *1* that represents the membership degree of each pixel to the category (either burned or unburned): *0* means that the pixel is more likely to be Unburned (U), *1*, instead, Burned (B).

c) Ordered Weight Averaging (OWA) computation:

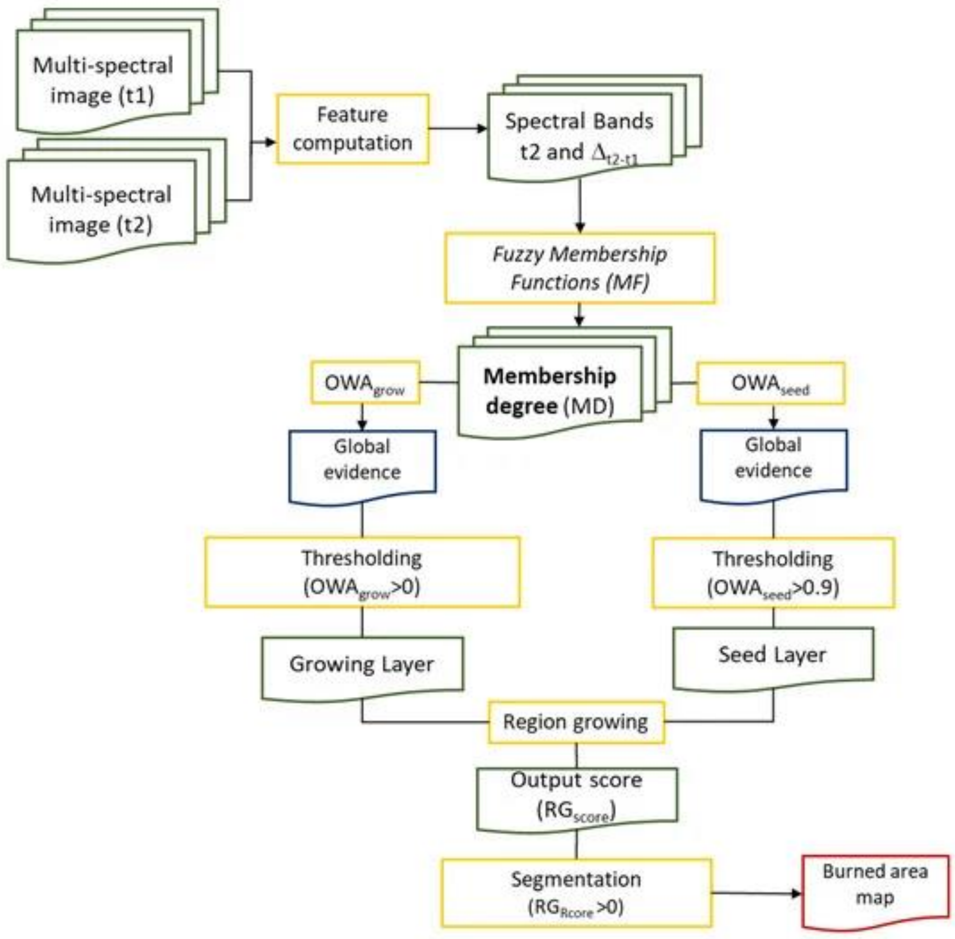
OWA operators are exploited to integrate MDs from different features into a synthetic score; these operators implement the convergence of evidence (agreement from redundant information/observations provided by the inputs). The synthetic score can then be converted to binary information Burned/Unburned, due to the fact that the final result should be a binary map in which each pixel contains only one information: Burned or Unburned. Indeed, up to step b), it is computed a continuous value of MD for each pixel and input feature (*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 features. *OWA* operators 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 the area burned: 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 the area burned: modeling a concurrent aggregation to integrate mutual reinforcement of multiple criteria).

Layers of global evidence derived with different $OWAs$ will be input to the region growing algorithm and provided to in this project by IREA CNR. The raster file (*.tif*) are subset to a smaller size (1400 by 1400 pixel) for faster implementation of the developed code.

d) Region Growing:

this last step is based on two different layers: *Seed* and *Grow*. The *Seed layer* in a RG algorithm represents the first initial partition of the image that is composed of those pixels that are most likely to be burned. In the fuzzy BA algorithm, the seed layer is generated by computing the OWA_{and}, and it is fundamental because allows to reduce the commission errors. The *Grow* layer is used to iteratively expand seed pixels; the *grow layer* is obtained from the OWA_{or}, and it allows to reduce classification errors by adding pixels that are less likely burned only if they are neighbors of the more likely burned pixels (seeds). These two layers combined in a contextual region growing algorithm can provide the most accurate results in terms of burned area mapping. As the name suggest the basic idea is to make grow the *grow layer* on the seed pixels (that belong to the *seed layer).*



Multi-spectral image (t1)
Multi-spectral image (t2)
Feature computation
Spectral Bands t2 and Δ_{t2-t1}
Fuzzy Membership Functions (MF)
Membership degree (MD)
OWA_{grow}
OWA_{seed}
Global evidence
Global evidence
Thresholding (OWA_{grow}>0)
Thresholding (OWA_{seed}>0.9)
Growing Layer
Seed Layer
Region growing
Output score (RG_{score})
Segmentation (RG_{Rcore}>0)
Burned area map

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

2. Implementation

This paragraph focuses on the implementation of step d) of the previous section in Python starting from the *OWA* layers provided as input in raster (.tif) format.

So, the first step is the computation of the *seed* and *grow layers*. The first one is obtained after the discretization of the OWA_{and}: if the pixel OWA_{and} value 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 step, that it is done 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 corresponds to an image pixel.

The next step in the RG algorithm is to implement the iterative growing process that considers one pixel at a time to select seeds from those matrix elements in the *seed layer* that are equal to *1;* afterwards, the algorithm checks the eight neighbor pixels of each seed to verify if the growing condition is satisfied (*OWAgrow >0.1*). Those neighboring pixels that satisfy the condition are selected and become new seeds. This procedure is repeated until no new seeds are found or the border of the matrix is reached.

Two approaches are tested to implement the RG code and to optimize computational time:

- scrolling the matrix Pixel by Pixel (*PixelbyPixel*);
- summing the layers (*LayerSum*).

In this project, the two approaches, coded in Python, are tested for execution time to identify the most efficient one for the implementation of the RG algorithm.

2.1 Data structure

Python offers the possibility to create multidimensional array (to be more precise it will be used a multidimensional matrix) thanks to the *numpy* library. In this project, the input multidimensional array is organized as follows (*Figure 2*):

- the first matrix contains the seed layer (*OWAseed*);
- the second matrix is the grow layer (*OWAgrow*);
- the last matrix (*T*) is an index term, that will be useful during the implementation of the iterative processing. *T* is initialized at the value of *-1*.

The structure is the following one:

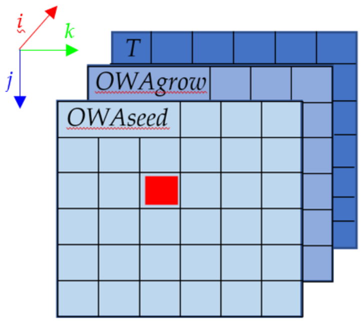


Figure 2. Data structure

Therefore, with the term *i* identifies one of the three matrices (*i* = *0*, *1*, *2*) (notice that Python start counting from *0*). Instead, with *j* and *k* is identified the cell of the matrix (*k* = row and *j* = column).

For example, looking at *Figure 2,* the red cell can be access using the following sequence: [*i*][*band*,*j*,*k*] = [*0*][*0*,*2*,*2*]. Once the data input to the RG algorithm is prepared, the algorithm is implemented with the two approaches.

2.2 Scrolling the matrix Pixel by Pixel

The first approach is based on scrolling through the *OWAseed* matrix thanks to two '*for*' cycles (appendix A code's *line 15-16*): seeds are identified as elements with value *1* (aA *l.20*). At this point, when a seed is found, its eight neighbors are analyzed (aA *l. 26-81*). Some conditions are added 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 neighbors pixels/elements, if it is not already a seed (*OWAseed = 0*), values inside the grow matrix are considered (aA *l. 26-81*).

If the condition (*OWAgrow = 1*) is satisfied, the element is added to the seed matrix, and, before proceeding, its state (matrix *T*) is modified (inside each '*if*' condition).

Thanks to this shrewdness, it is possible to know at which iteration a pixel/element is inserted as new seed: it is sufficient to update the value initialization 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 will not be considered again.

However, updating matrix T at each step increases execution time of the RG algorithm. In order to reduce execution time, the computation of *T* could be removed without compromising the effectiveness of the algorithm. In fact, without storing the number of iterations in matrix T, it is not more necessary to read and write/overwrite in memory the matrix (this improvement is shown in a graph in paragraph *3*).

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

2.3 Summing the layers

The second approach proposed in this project to implement the RG algorithm 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*, able to identify the condition "*being a neighbour of a seed pixel*". In order to perform this operation, seeds are selected from the *seed matrix* and the eight neighbors of each seed are set to *1* with Py function: *CercaVicini*.

From the article [2] an assumption may be derived: if a pixel satisfies the condition OWA_{and} = *1* (it is a seed), then for sure it satisfies also the condition OWA_{or} = *1*: that element will be equal to *1* in both layers.

This assumption becomes of key importance for this method, but it is better to highlight the fact that it can be considered with the approach proposed by the article [2] and it is a consequence of the definition of the *OWAs* that generated the seed and grow layers. It is not a general hypothesis for RG algorithms, so in any other case it should be verified.

So, there are two values (*0* and *1*) that describe three scenarios in the input layers (*Figure 3*):

- *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).

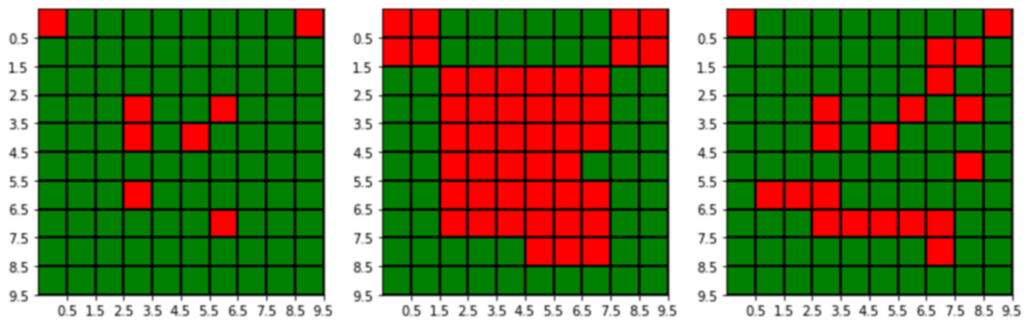


Figure 3. In order: seed matrix, neighbor matrix, grow matrix

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

- *0* means that element 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 it is a neighbor for a seed (red);
- *2* means that element is burned for both layers 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*).

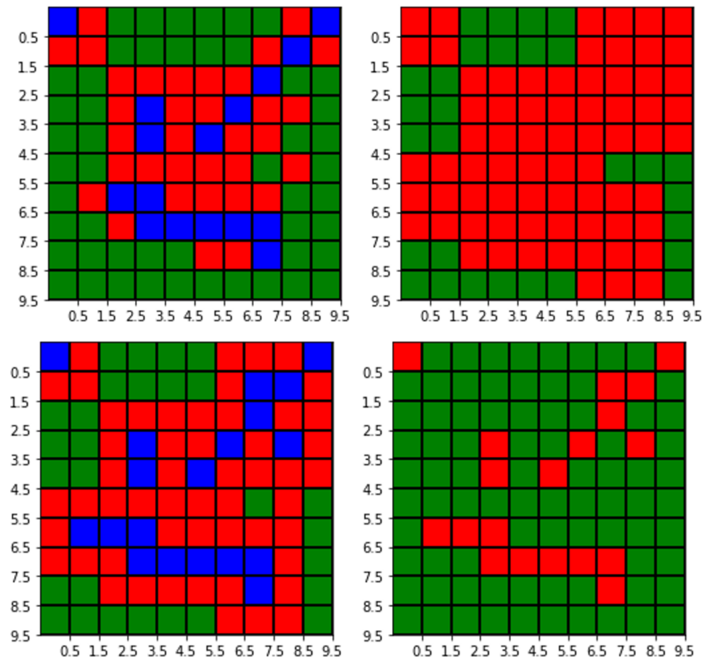


Figure 4. In order: sum at cycle 1, neighbors and seed at cycle 2, sum at cycle 2, final result

3. Test and Discussion

This section shows the results of the two approaches implemented in Python and analyzes some performance indicators to highlight whether one of the two approaches could be preferred especially for optimizing processing of large input images (large rasters, generally a tile from S-2 has dimension equal to 100 *km* by 100 *km*). The results presented in this section are obtained for a case study of Sentinel-2 images acquired over Portugal (pre-fire:4th June 2017, post-fire: 4th July 2017) and subset to a 1400 by 1400 pixel window, this window covers an area of approximately 14 *km* by 14 *km*, given pixel size of S-2 image of 10 m.

Results show that processing time is quite similar for the two approaches for small raster/matrices, but the first method (*scrolling the matrix Pixel by Pixel*) offers a faster solution when the image's size starts to increase (> 400x400 pixel, *Figure 5*).

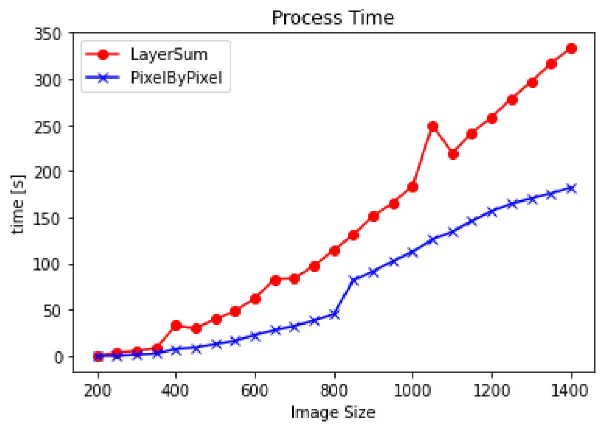


Figure 5. Process time [second] for Images with size that goes from 200x200 to 1400x1400 pixel (step 50)

As already anticipated in *section 2.2*, the execution time can be further reduced, as shown in *Figure 6*, by removing the computation of matrix *T*. In this case, if a neighbor pixel is recognized as a seed, the program goes ahead and does not evaluate it. *Figure 6* shows processing time for increasing window size of the *PixelByPixel* approach with (red) and without (blue) the computation of matrix *T*.

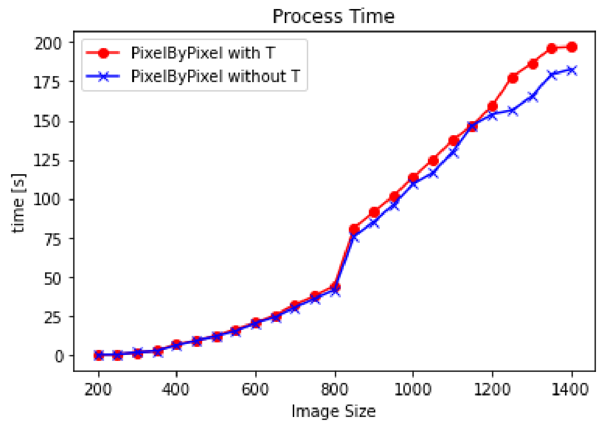


Figure 6. Process time using the first method with T and without on images with size that goes from 200x200 to 1400x1400 pixel (step 50)

As a second test, was analyzed the number of seeds for each iteration (algorithm convergence) and *Figure 7* shows the result for the two methods. Both approaches scroll the matrix at each cycle; however, the first method (*PixelByPixel*) 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 as new seed during the scrolling of the matrix rather than at the next iteration, thus saving time.

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

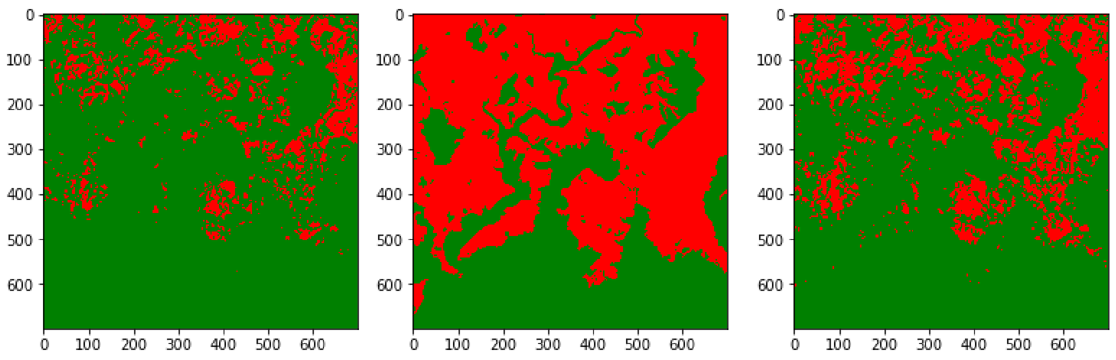


Figure 7. In order: seed layer, seed layer after one cycle with first method, seed layer after one cycle with the second method

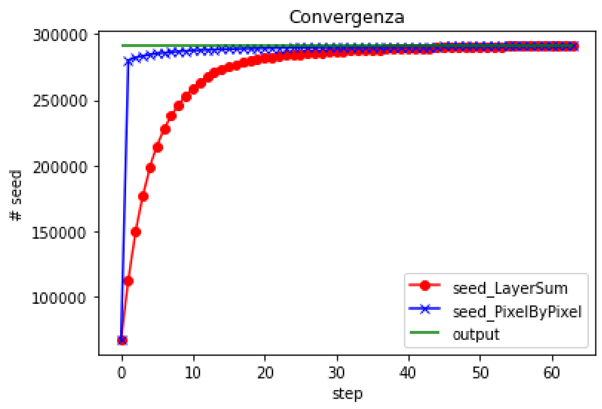


Figure 8. Number of seeds at each scrolling cycle of the seed matrix (image's size 700x700 pixel)

It was also tested if there is a relationship between processing time and spatial aggregation and initial number of seed pixels. In order to do this, the two approaches are applied to five random regions of 700x700 pixel (clipped from the input raster) where input seeds have different spatial distribution as highlighted in *Figure 9*.

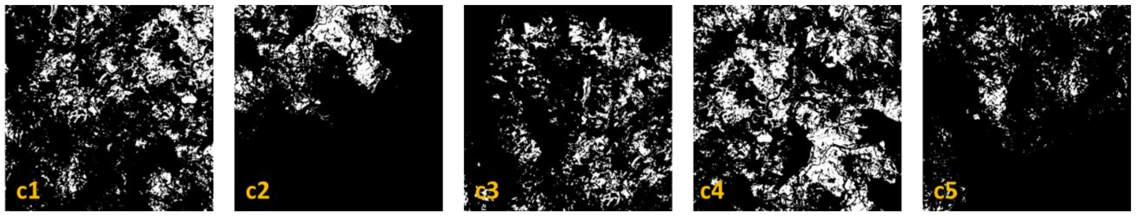


Figure 9. Clips of 700x700 pixel of seed layer. In white Burned pixel, in black Unburned ones

From the trend of the graph (*Figure 10*) it is not possible to conclude that there is a dependence between the two executions time and the number of initial seeds, but the time will depend also on how the seeds are distributed and clustered in the considered subset area. For example, looking at clip C$_2$, it has a high concentration of seeds in the upper left area and needs less time to analyze it, which is consistent with the lowest execution time shown in *Figure 10.*

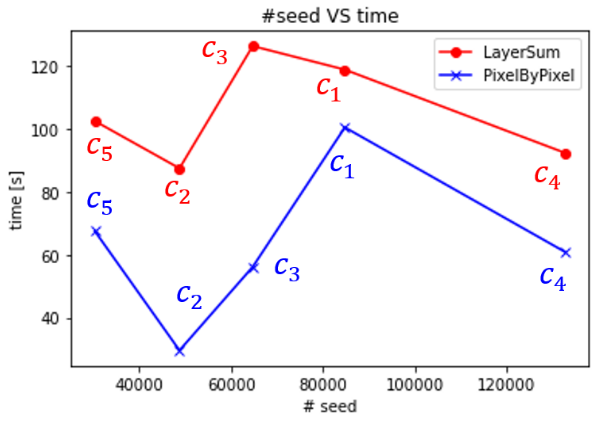


Figure 10. Process time with respect to each clip (c_i)

Finally, the result that are obtained with both methods are compared with the expected output (considered as reference), that was derived by appli the region growing algorithm in a different software (ENVI Exelis Visual Information Solution, Boulder, Colorado). Since the two approaches give the same results, for simplicity, it is reported one case (*PixelByPixel*) where the output from the Py RG algorithm developed in this project is presented for a 700x700 pixel window

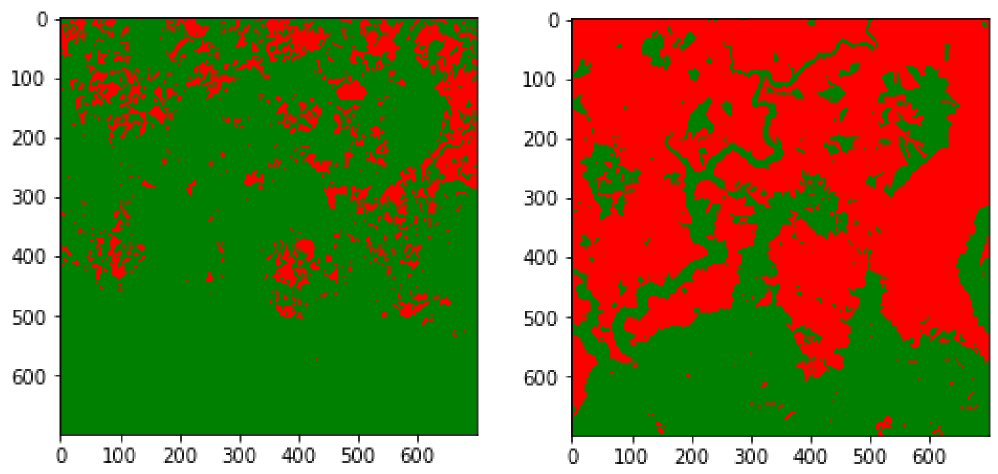


Figure 11. Example seed layer (left) and grow layer (right) for a 700x700 pixel window (unburned is green, burned is red).

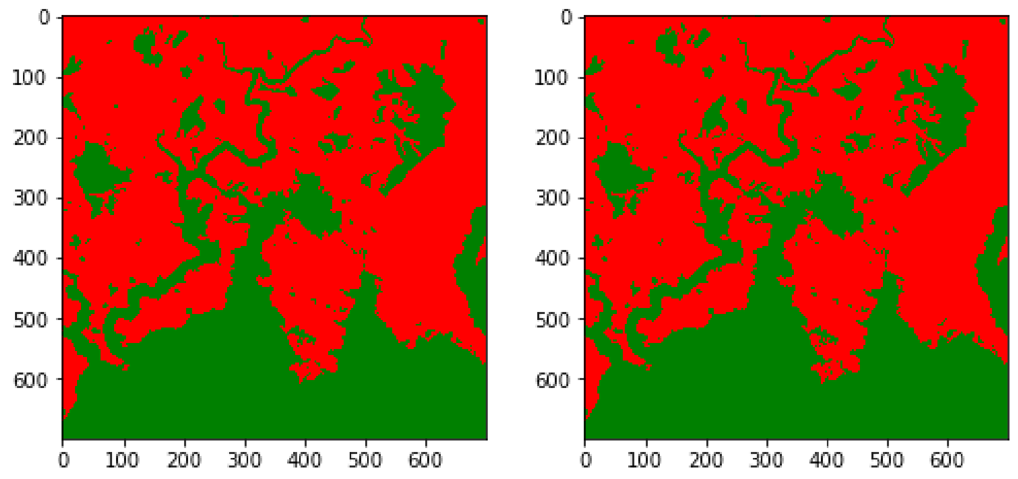


Figure 12. Example ENVI (left) and Py RG (right) output result

In the example figure shown in *Figure 12*, the output of the Py RG algorithm implemented in this project and the reference derived with ENVI software are very similar. If we compute the confusion matrix between the two outputs, we can estimate that most of the burned pixels are detected (99,65%) and only 0.57% of the pixel are erroneously assigned to type burned class (commission) and 0.91% are not detected (omission) as shown by the confusion matrix:

		REFERENCE			
		Burned	Unburned	Total	Commission
ALGORITHM	Burned	*289467*	*1647*	*291114*	*0.57%*
	Unburned	*2664*	*196222*	*198886*	
	Total	*292131*	*197869*	*490000*	
	Omission	*0.91%*			

4. Conclusion

In this project, a Region growing (RG) algorithm was coded in Python (Py RG) to be implemented in a fuzzy burned area mapping algorithm. The objective was to provide a Python code to be integrated in an open source algorithm for mapping the area affected by fires from Sentinel-2 multi-spectral satellite images. The RG code is implemented with two approaches (*PixelByPixel*, *LayerSum)* and tested for execution time to find the most efficient one for processing large raster images.

Results show that, although the *LayerSum* approach is more intuitive and immediately understandable, it is slower 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*. In fact, in the *PixelByPixel* approach, the *seed matrix* is immediately updated and information on the presence of new seeds is available in the next step.

Results from this project show that the *PixelByPixel* method can be further optimized by removing the computation of the *T* matrix, that takes into account the number of iterations.

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 on burned pixels after a few cycles. This would allow to analyse areas of greater extension in less time.

References

1. ESA, Copernicus, Sentinel-2 [Link]
2. 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]
3. Copernicus Schi Hub [Link]
4. IREA CNR [Link]
5. Documentation numpy [Link]
6. Documentation matplotlib [Link]
7. Documentation rasterio [Link]
8. Documentation os [Link]

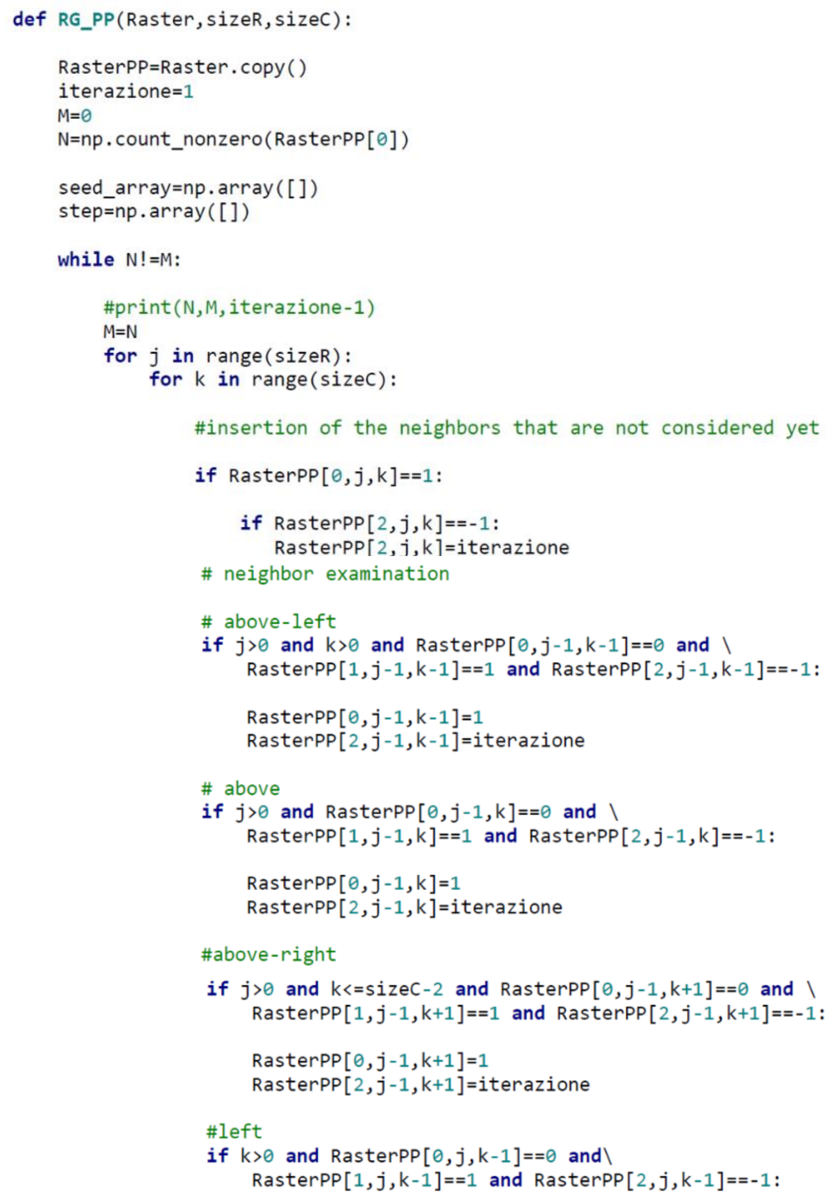
Appendix A

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.



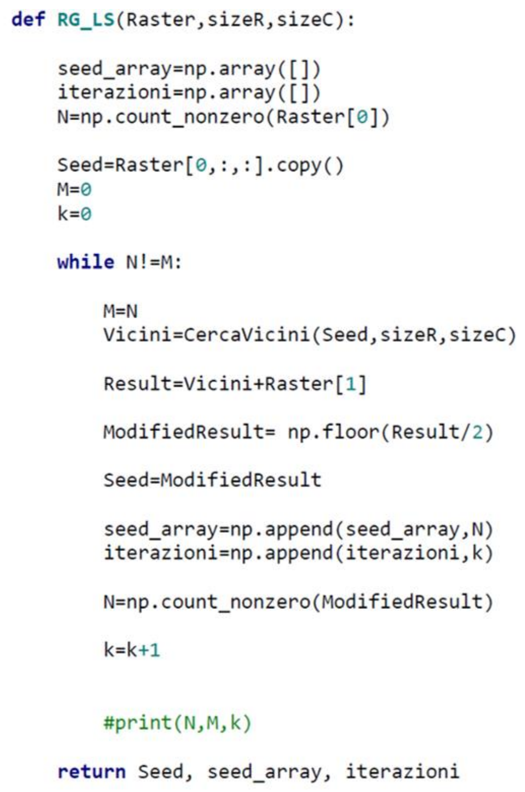
```
1    def RG_PP(Raster,sizeR,sizeC):
2
3        RasterPP=Raster.copy()
4        iterazione=1
5        M=0
6        N=np.count_nonzero(RasterPP[0])
7
8        seed_array=np.array([])
9        step=np.array([])
10
11       while N!=M:
12
13           #print(N,M,iterazione-1)
14           M=N
15           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                    #above-right
41
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
```