UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Data processing and graph visualisation of tumour and t-cell detections

A thesis submitted for the degree of

Bachelor of Computer Science

By

Yannis Laaroussi

Supervisor: Linda Studer

University of Fribourg
DIVA research group
1700 Fribourg
Switzerland

September 2022

Abstract

The detection of tumour and t-cell is a very important topic in medicine. It is useful to identify colorectal cancer. As computer scientist our goal is to provide some applications in order to help people working on it. The project intends to develop some helpful scripts. There are two different parts about these, data processing and visualisation. Data processing consist in making some conversions from a data type to another one. The goal is to make data more readable and usable in other programs. Visualisation is achieved by drawing of graphs on hotspots images showing a specific area containing tumour and t-cell. Another purpose of visualisation is to implement a dynamic interface where a user can directly interact with it.

Keywords: Tumour/t-cell, data processing, graph visualisation.

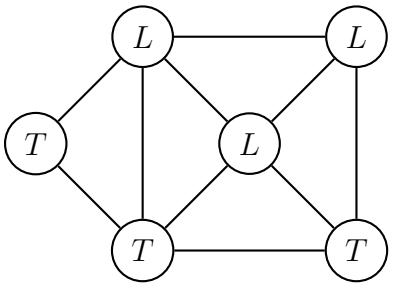
Contents

1 Introduction

This thesis is talking about some applications that are useful for medicine purposes. Especially the detection of tumour budding and t-cell in order to identify a potential colon cancer. It is also related to the PhD Thesis of Linda Studer [1]. On the article [2] it explains how these cells work in colorectal cancer. Especially for doing some prognostics in order to treat correctly the patient.

We have a lot of data in image format coming from a fixed incisions of colon tissue. From these ones we can extract another type of data in a textual format (precisely the json format) that contains annotations and also other features. The aim of this project is to use these data (images and annotations) for some applications. It is divided in two parts, data processing and graph visualisation. The first part, data processing mainly focus on the conversion of a data type to another one. There are three conversions in this project two of them are about passing from json to xml file [3] and the last one is from json to csv file. The primary goal of that is to make data usable for different software which is not the case with the initial data. The software that use the new processed data are useful to visualize better the data than if it was only in json format.

The second part is about graph visualisation. The basic idea is to draw a graph [4] on a hotspot image. The goal is to represent the structure of the different cells with a graph. This approach is called histocartography, described in [5] and [6]. Schematically and in a simple way a graph would look like this:



Here, the nodes "T" (resp. "L") are used to represent tumourbud (resp. Lymphocyte). Another purpose is to provide a dynamic graphical interface. With it, the user would be able to customize the graph as choose the color of the nodes, size of node, modify transparency, etc. and can directly observe the changes on the screen.

There are a plenty of way to implement the different applications. For this project we use the programming language Python. It is one of the most use language and also very convenient due to the large amount of libraries available. With them it is possible to process easily different type of data (e.g. open, read, write, etc.).

2 Data Overview

This part focus on the description of the data that will be used in the different applications describe at section 3 and section 4.

2.1 Images

There are many types of images which all come from fixed incisions of colon tissue. After an incision, the tissue is colorized in order to distinguish the different types of cell and then scanned to visualize it on the computer.

2.1.1 Whole slide image

The whole slide image [7] is the main object, see Figure 1. This is the largest type of data (its size is between 90'000 and 200'000 pixels) that we have for this project and it is saved in *mrxs* format.

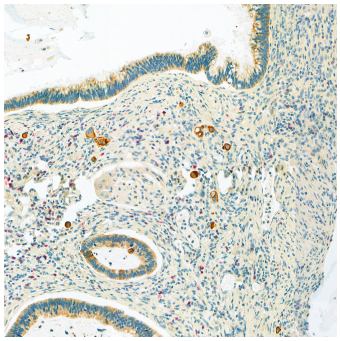
Figure 1: Whole slide image

The *mrxs* format is useful because it allows us to load the image in the software **ASAP** [1]. It is a really good software to visualize images and also own several annotation tools integrated. It also has the functionality that we can load a *XML* file containing annotations and directly draw it on the picture.

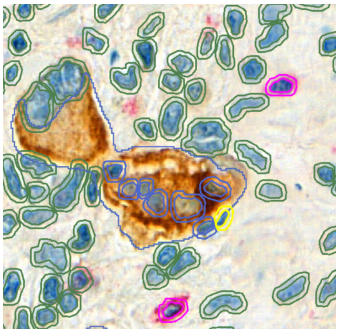
[1] website: `https://computationalpathologygroup.github.io/ASAP/`

2.1.2 Hotspot

Hotspot images are specific areas in the whole slide image, hence the size of them is strictly smaller (3'600 $\times$ 3'600 pixels) and there are about 340 images of this type. This is time the format is *png* because it is required to load it on the software **QuPath** [2], see the article [8] to have more information about this software. With it, we can see the boundary of every cells, which are called annotations. For example, on the Figure 2b, which is a specific location of the Figure 2a, the brown cell is a tumour bud and is bounded by a blue polygon.



(a) Hotspot image $3600 \times 3600px$



(b) Hotspot image open on QuPath

Figure 2: Hotspot images

2.1.3 Tissue micro array (TMA)

The last type of image that we have is a kind of combination of the two previous one (i.e. whole slide and hotspots images), which is called Tissue micro array [9]. This is also a *mrxs* image as Figure 1. The hotspots have circular shape and are placed line by line, see below (Figure 3).

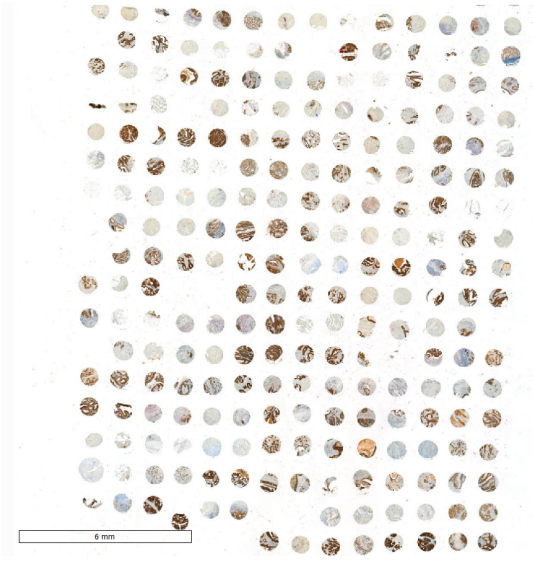


Figure 3: TMA

[2] website: `https://qupath.github.io/`

2.2 Annotations

Available data are not only images but also extra information are in textual form. For each hotspot images, there is a *JSON* file associated that contains the following features:

- Area
- Center of Mass
- Centroid
- Circularity
- Classification (e.g. tumor, lymphocyte, ...)
- Solidity
- Minimum and maximum diameter
- Number of cells
- Object Index (ID)
- ROI points (= region of interest points, i.e. the coordinates of the polygon that delimit the cell).

JSON is are very good format to store data because it is easy to read and use. A lot of programming language have the capacity to process this type of file (e.g. using a library with Python or Java). Other data are on the *XML* and *CSV* form but it only contains coordinates that are used for positioning the hotspot annotations at the right place on the whole slide image (Figure 1) and also on the Figure 3.

3 Data Processing

3.1 Problem statement

Data processing consist of making some conversions, mainly from the *JSON* file described in 2.2 to other format as *XML* and *CSV*. There are three different conversions based on the different images available:

1. **JSON** (+ XML) → **XML**: The goal is to use the *JSON* file of each hotspot image in order to convert it into an *XML* file that can be read by **ASAP**. This software draws the annotations of the hotspot (Figure 2) on the whole slide image (Figure 1). For the coordinates of the annotations to be at the right place, it also needs another file that contains the coordinates of the hotspot on the whole slide image. This extra file is of *XML* type.

2. **JSON** → **CSV**: In the previous point, it only uses the ROI (region of interest) points of the *JSON* file. While this second conversion is to use the one dimensional other features (e.g. Area, Circularity, Classification, Solidity, ...) and store them in a *CSV* file in order to open data on excel.

3. **JSON** (+ CSV) → **XML**: This last conversion is similar to the first one but this time it intends to convert the *JSON* file of each hotspot image into an *XML* file that can be use for the Figure 3. A *CSV* file is available to update the coordinates of the hotspot annotations to have each annotations on the corresponding ”circle”.

3.2 Strategy

As mentioned before in the chapter 1, all programs are implemented in Python. The main strategy consists of using libraries offered by the language. First of all, for the three conversions, this is necessary to read *JSON* files, hence we must have tools to deal with it. Python owns a standard library called simply **json**, which allows to open, read and write *JSON* file. This is also necessary to be able to process *CSV* and *XML* files. For *CSV* file. Again, there is a standard library called **csv** with mainly the same properties as the **json** one. Meanwhile there are several libraries to process *XML*, lxml.**etree** [3] library is useful to create xml tree and the xml.dom.**minidom** [4] library for the opening and reading.

For the user to give the desired files as input, the library **argparse** perfectly fit this purpose. This one allows the user to pass the directory where the files are located (or directly the file path, it depends on how this is implemented/what is required) in a command line. The library **os** is also crucial to access and navigate with the different file/directory path giving by the user.

The last useful library is **re** that is a very interesting one because it allows to create regular expression. With it, we can check for files to take/ignore in a specified directory.

[3] documentation lxml: `https://lxml.de/tutorial.html`

[4] documentation minidom: `https://docs.python.org/3/library/xml.dom.minidom.html`

3.3 Implementation

3.3.1 Argument parser

First thing to implement is the argument parser that allows the user to pass the files through a command line. With the library **argparse**, we can add arguments that force the user to fill them, otherwise the program won't be executed. Let's look at an example:

```
parser = argparse.ArgumentParser()
parser.add_argument('--input_file', type=str, required=True)
args = parser.parse_args()
```

Here we create an new parser and we add the argument *input_folder* with the attribute "required" set to true and of type string. It means that the user must pass a string for the argument *input_folder*. We can run the code by opening a shell in typing for example this commande line:

```
python example.py - -input_file exfile.json
```

The variable **args** is necessary to get the value given by the user (i.e. *exfile.json* in the example above).

Second step is to check the folder that the user pass in the command line. This is done by the usage of regular expression. For the three conversion it has to pass a directory containing json files. Then we need a regular expression that check in the folder which one are of type json and which are not. Additionally it should not take into account the json files that have not a valid name (e.g. a valid file containing data about hotspot can have the name *Masks_00.2205_1D_AE1_AE3_CD8-level0-hotspot.json*, hence if there is another file in the folder having the name *birds.json*, it should not be considered because the name is clearly not the accepted format). With **re** [5] we can create a regular expression that can accept the following skeleton:

Masks_[A-Za-z0-9]-level0-hotspot.json*

This regex means that the string must start with *Masks_*, then any characters (0 or more) and ends by *-level0-hotsport.json*. We can implement it by using the search or match method. For example, with search method we give as input the regex expression and the test string:

```
re.search('Masks_.*-level0-hotspot.json',
'Masks\_00.2205\_1D\_AE1\_AE3\_CD8-level0-hotspot.json')
```

In this example, the result will be true as the test string is validate by the regex (note that: **.*** means anything). Additionally, the papers [10] and [11] are good for a better understanding of regular expression.

[5] documentation re: `https://docs.python.org/3/library/re.html`

3.3.2 XML format processing

For the conversions 1 and 3, the output is an xml file that can be read in **ASAP**. Then for the both conversions, the goal is to construct an **xml tree** (see [12] to have more details about xml tree and how useful they are). In order to make the xml tree readable by **ASAP**, it must have the following shape:

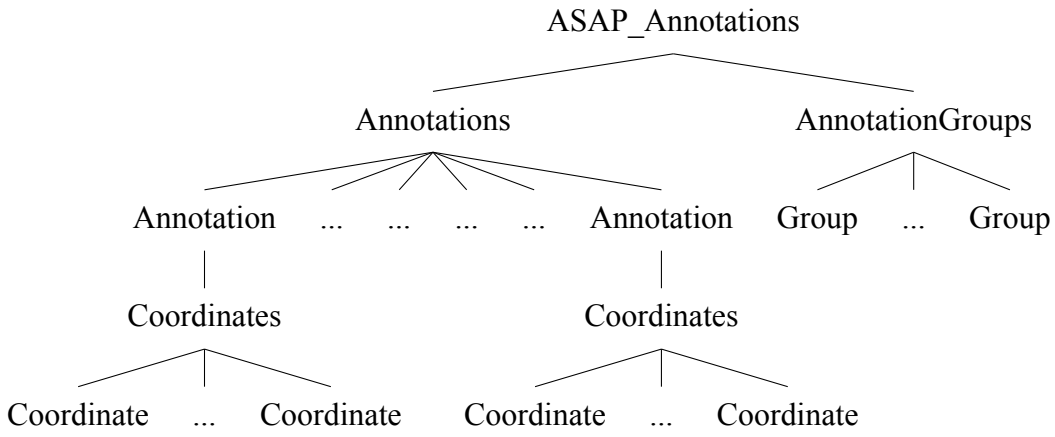


Figure 4: xml tree readable by ASAP

The root note is **ASAP_Annotations** and it is divided into two main branches which are **Annotations** and **AnnotationGroups**. The first one contains the coordinates of each annotation (i.e. coordinates of the polygon that delimit the cell) and also indicate in which group the annotation belongs. The second one deals with the different groups, which are useful to classify the annotations. A group contains a name (e.g. Tumor, CD8+ Cell, Center of Mass, ...) and a color (i.e. the color with which the polygon is drawn on ASAP).

The implementation of Figure 4 with lxml.**etree** (renamed **ET** in the code) basically works as follow: We create nodes from top to bottom (root to leaf). More precisely, we start by creating the root node **ASAP_Annotations** with the Element method:

```
root = ET.Element('ASAP_Annotations')
```

For the creation of all the branches, we use the SubElement method that takes the parent and the name of the new node as parameter. Then for the branches **Annotations** and **AnnotationGroups**, this is implemented as follow:

```
annotations = ET.SubElement(xml_tree, 'Annotations')
annotation_groups = ET.SubElement(xml_tree, 'AnnotationGroups')
```

Additionally, some nodes own attributes, e.g. the **Coordinate** nodes must contain the coordinate of a ROI (region of interest) point (i.e. a point that builds up the polygon). Here we just have to pass the attribute which is a dictionary to the method SubElement. For example, the first point of the polygon is at the position $(100, 200)$:

```
coord_attrib = {'Order': '0', 'X': '100', 'Y': '200')}
coordinate = ET.SubElement(coordinates, 'Coordinate', attrib=coord_attrib)
```

The key *Order* means the current point number, i.e. **ASAP** draws a polygon using the ascending order. It draws lines from 0 to 1, 1 to 2, ..., n-1 to n, n to 0 (for a polygon with n + 1 points).

For conversion 1 and 3, this is also necessary to place the coordinates at the right place on the whole slide image and TMA (Figure 1 and 3) because the coordinates of the ROI points stored in the json file only give the position on the hotspot image (Figure 2). For the first conversion (1), there is an xml file containing the coordinates of the hotspot. The structure is the following:

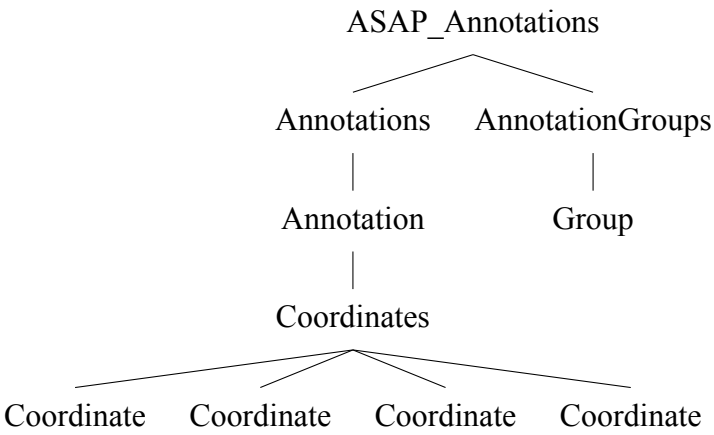


Figure 5: xml tree for the hotspot coordinates

As we can observe, this is the same structure as Figure 4 but this time the xml tree only contains coordinates of the hotspot (4 coordinates because this is a square) and has only one group (hotspot group).

Using **minidom** we can extract the values of the elements **Coordinate** with the help of *getElements-ByTagName* method. Then we can easily get the attribute values:

```
1    hotspot_file = minidom.parse('coordinates.xml')
2    coordinates = hotspot_file.getElementsByTagName('Coordinate')
3    hotspot_coord = [(float(point.attributes['X'].value), float(point.attributes[
    'Y'].value)) for point in coordinates]
```

As soon as we have the coordinates of the hotspot we can write them onto the xml file and move the ROI points inside of it.

3.3.3 CSV format processing

For the third conversion (3), there is a csv file containing coordinates of the hotspot. It has the following structure:

	A	B	C	D
1	Core Unique ID	Centroid X (pixels)	Centroid Y (pixels)	Radius (pixels)
2	1651	14667	14511	1621
3	1650	19005	14980	1621
4	1649	23472	15172	1621
5	1648	27830	15697	1621
6	1647	31930	16068	1621
7	1646	36359	16171	1621
8	1645	40788	16171	1621
9	1644	45114	16377	1621
10	1643	49646	16480	1621
11	1642	53972	16583	1621
12	1641	58298	16686	1621
13	1640	62933	16377	1621
14	1639	67156	16583	1621
15	1638	71585	16686	1621
16	1637	13287	18715	1621

Figure 6: csv containing hotspot coordinates

There are four columns, the first one is for the ID, which identify the hotspot. The columns B and C indicate the center of the circle (see Figure 3, hotspots are not square anymore). The last one is for the length of the radius. Now with **csv** library we can easily open and read the values above. In order to have the appropriate coordinates for the ROI points, we only have to add the center coordinates and subtract the radius.

For the last conversion (2), the strategy for saving features into a csv file is to create a dictionary for each row.

```
    with open(output_file, 'w', newline='') as csv_file:
        writer = csv.DictWriter(csv_file, delimiter=';', fieldnames=features)
        writer.writeheader()
        for obj in objects:
            row = {feature: obj[feature] for feature in features}
            writer.writerow(row)
```

Firstly we open the output file and set up for writing with dictionary. Then for all json files we extract the interested features and save them on a dictionary that is used to write a row in the csv file.

3.4 Validation

For the first conversion (1), as explained before, the xml file generated by the script can be loaded into **ASAP**. Hence, we can easily check for the validity of the application by loading the file to the corresponding whole slide image (Figure 1).

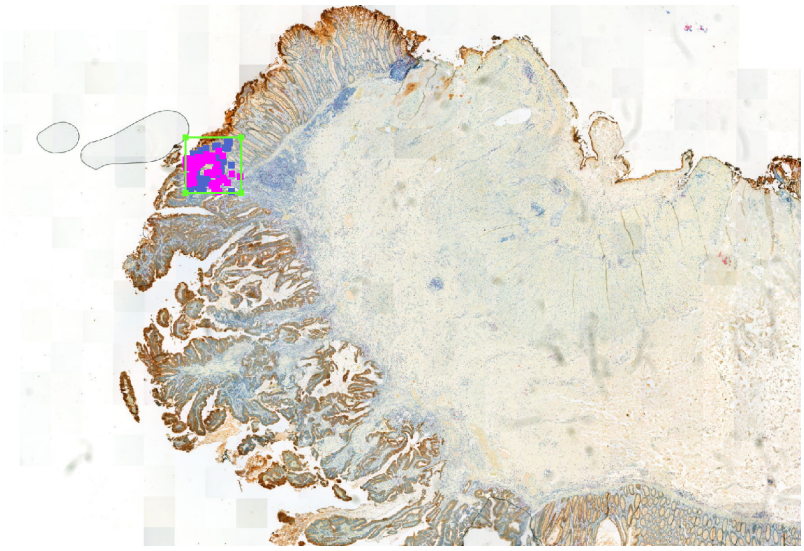


Figure 7: Whole slide image with the drawing of the hotspot

For the second conversion (2), a csv file is created with all the one dimensional features and in order to check the validity, the file is opened onto Excel.

	A	B	C	D	E	F	G
1	Object_Index	Classification	Area	Circularity	Number_Cells	Perimeter	Solidity
2	0	Tumor	5976	0.2614	1	535	0.7297
3	1	Tumor	7336	0.1319	1	836	0.5314
4	2	Tumor	2204	0.2396	1	340	0.7512
5	3	Tumor	2368	0.4686	2	251	0.9171
6	4	Tumor	4660	0.5719	1	319	0.9581
7	5	Tumor	1660	0.4471	1	215	0.9012
8	6	Tumor	14076	0.1842	5	980	0.7237
9	7	Tumor	5048	0.1886	1	579	0.666
10	8	Tumor	11948	0.1643	1	955	0.6004

Figure 8: csv output file loaded on excel

The table above is a very useful tool. For example, we can directly make plots or other statistics stuff (e.g. diagram) on Excel.

The last conversion works on the Figure 3 with the same principle as the first one. When loading the xml output file, **ASAP** draws the annotations on the corresponding hotspot.

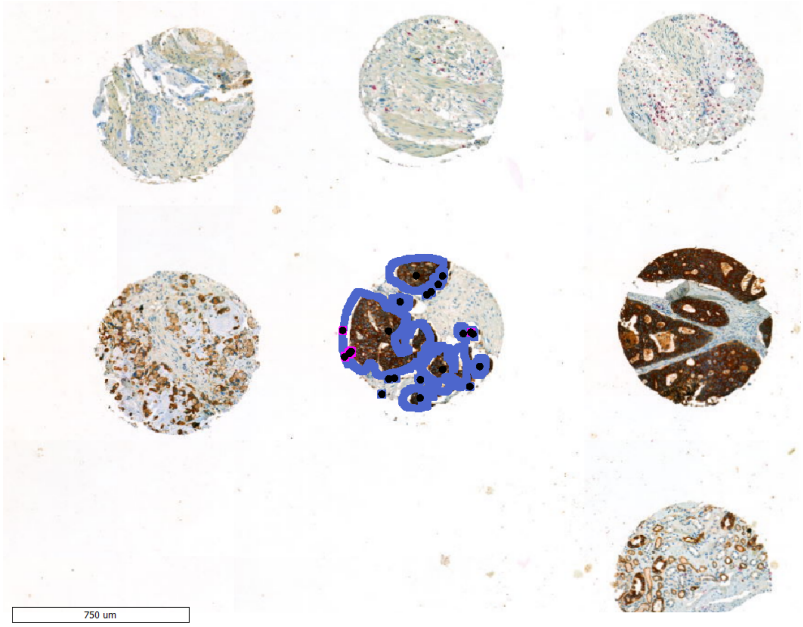


Figure 9: Drawing of the circular hotspot

4 Visualisation

4.1 Problem statement

The visualisation part is about to draw graph above an (hotspot) image. The goal is to provide a dynamical interface where the user can directly interact with it. The main part is to allow user to load an hotspot image and another file containing graph coordinates. This other file is of *gxl* type [13], which is a similar format to xml but it is most suitable for storing graph coordinates. Hence it is also possible to represent the structure of the *gxl* file as a tree.

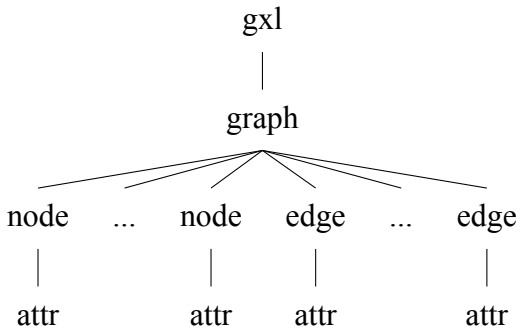


Figure 10: gxl tree

Every graph is composed with nodes (points) and edges (line between two nodes). On the Figure 10 every node element has an ID and every edge contains the ID of two nodes. In the attribute leaves (attr) of the nodes, there are the coordinates (x,y) stored and can also contain extra attributes as the type of a cell (e.g. 'tumorbud'). For the edges, the attribute leaves own the distance between the both nodes composing the edge.

Overall, the application has the following features :

- Load folder (image and gxl)
- Draw graph
- Change color of nodes and edges
- Edit transparency
- Edit scaling
- Save images

These features must be dynamic, i.e. the user can see the current state of the image (graph + hotspot) without have to save it in a file. For example, if the user change the color of the edges, the application directly updates the graph on the screen.

4.2 Strategy

Similar to the strategy of the data processing part (section 3.2), the visualisation part also needs libraries as **os** (for the processing of every file/folder path for input/output), **re** (find the proper gxl file corresponding to the given image) and **xml.etree** for the parsing of gxl file. Additionally it needs new libraries as **OpenCV** [6] and **matplotlib** in order to draw graphs on images. Now for the creation of the graphical user interface, the core library is **tkinter** [7]. With this tool, this is possible to create buttons, scrollbars, display picture in a canvas, etc. A functionality of this library is to place the elements on a grid, e.g.

button1	button1	label	...
button2	picturebox1	picturebox1	...
scrollbar	picturebox1	picturebox1	...
...	...	...	...

Figure 11: tkinter grid

Note that an element can take more than one square of space. In the example above, *button1* is placed on the first row and fill the first and second column. It is also possible to have unused squares.

4.3 Implementation

4.3.1 Open files

The first part of the implementation is to allow the user to be able to pass an image and gxl files folder. In a second time, we must check which gxl file match with which image. Firstly we create two buttons, one is dedicated for the image folder path and the other one for the gxl folder path. With **tkinter**, this is possible to create a button as follow:

```
1     img_dir = StringVar()
2     ttk.Button(mainframe, text='Image Path:', command=select_img_dir).grid(column
     =0, row=0, columnspan=2)
```

Every button is linked to a method that is called when the user click on it. Here the method name is *select_img_dir* and its job is to ask user what is the folder path and update the variable *img_dir*, which is storing the given path. We can also notice that the position of the button on the grid is (0,0) with a columnspan of 2. It means that the position is located at the top left and takes two columns of space (same as *button1* on Figure 11) .

For the matching of gxl file and image, again we need the usage of regular expression as seen in section 3.2. The idea is to display the gxl files as a list where the user can select one file and the application will draw the corresponding image on the picture box. In order to find the corresponding image of a gxl file, we firstly extract the name of the gxl file. Then we search for the image name that contains it. Formally, the regular expression is constructed as follow:

$$[A\text{-}Za\text{-}z0\text{-}9]^{*}gxlfilename[A\text{-}Za\text{-}z0\text{-}9]^{*}$$

[6] documentation OpenCV: `https://docs.opencv.org/4.x/index.html`
[7] documentation tkinter `https://docs.python.org/3/library/tk.html`

4.3.2 Drawing graphs

The second part is to draw the graph on the corresponding image. First of all, the user has to set a configuration, i.e. the size, color of the nodes and edges. Then we need to create an interface for this purpose. For the choice of a color, **tkinter** provide a very nice tool that is the *colorchooser* method. It can ask the user for a color by displaying this interface:

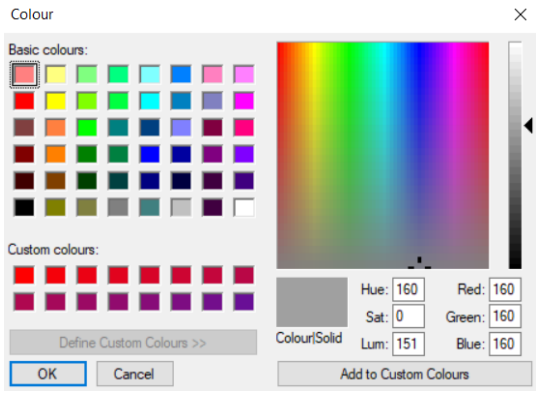


Figure 12: colorchooser

When the user select a color, the returned value is in RGB format and even in HEX code.

For the user to choose the size of node (radius) and edge (thickness), this is simply done by adding text fields where user can type the desired value. Additionally an interesting functionality is whether the user doesn't type a number, the application must erase what the user wrote and restore the previous value.

Now that the basic configuration is in place, this is the time to draw the graph on the image. As said in section 4.2, this is done by the usage of the library **OpenCV** (cv2). The first step is to create a class where we pass the image file, gxl file and the configuration. With the gxl file we get the coordinates of every node that we can use to draw the nodes and edges. The drawing of the nodes works as follow:

```
1    img = cv2.circle(img, point, radius=self.node_style[feature]['radius'],
2                                 color=self.node_style[feature]['color'],
3                                 thickness=self.node_style['thickness'])
```

First of all, the variable *img* is containing only the image without any drawing on it. Then we draw a circle for a node and not only a point because the user can choose the size of the node by modifying the radius. The variable *point* is a tuple with the (x,y) coordinates of the center of the circle. They are extracted from the gxl file. Finally the last arguments are for the configuration given by the user (color, radius, thickness).

For the drawing of the edges, this is quiet similar to the nodes. This time we need to have the coordinates of the two nodes composing the edge and then draw a line.

```
1     img = cv2.line(img, pt1, pt2, color=self.edge_style['color'], thickness=self
  .edge_style['thickness'], lineType=self.edge_style['lineType'])
```

Assume that we draw all the nodes first, hence the variable *img* will now contain the initial image and all the nodes above it. Here *pt1* and *pt2* are tuples with the (x,y) coordinates of the first and second node composing the line. As for the node, the last arguments are for the configuration given by the user, i.e. the color and thickness.

Another functionality to have is to allow user to modify the transparency of the hotspot image. For this purpose, we need to use the color on the **rgba** (red green blue alpha) format. The last parameter, alpha, is the one which indicates the transparency. It takes a value between 0 and 255 (as the other parameters) where the lowest value is for a total transparency and the highest one for a maximum of opacity.

4.4 Validation

The application works as follow: Before the user can interact with the different customisation features as modify color, transparency, etc. he must firstly pass the gxl and images folder as input with the two first buttons on the top left of the interface. Then the user can select a gxl file in the list box on the left and if a corresponding image match with this file, the program will draw the image and the graph on the canvas. Otherwise a text message is written on the canvas informing that no image match the selected gxl file. Afterwards this is possible to use the customisation panel on the right of the canvas. In practical the application looks like:

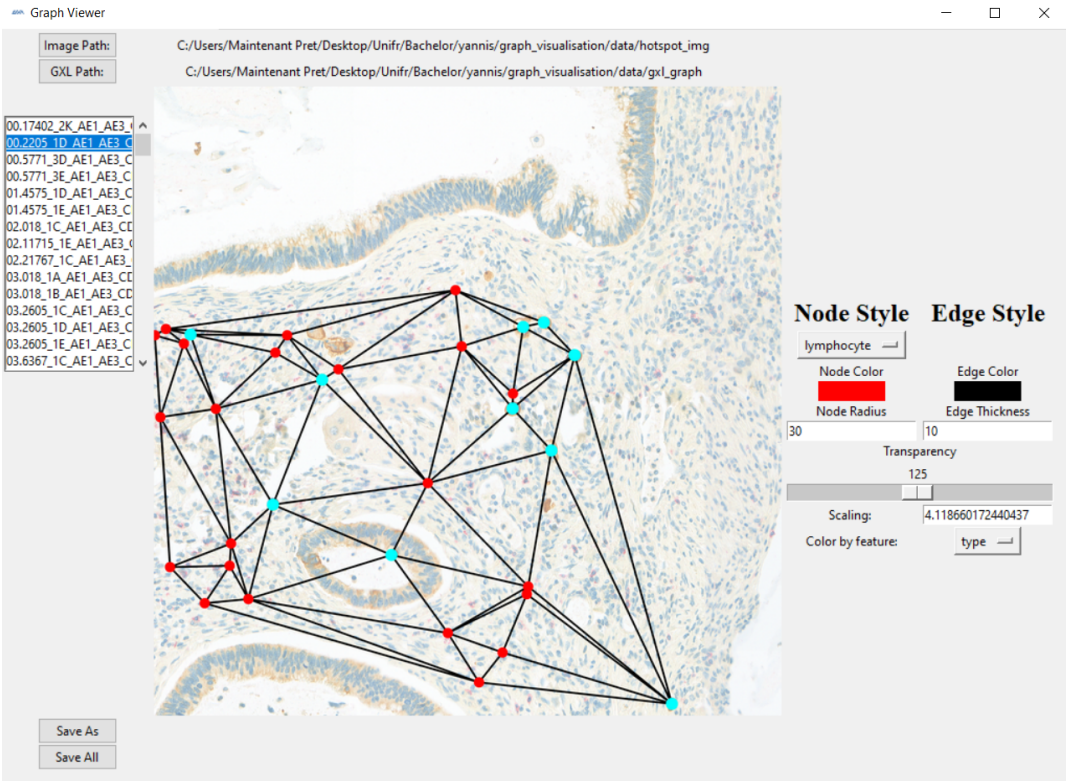


Figure 13: Graph visualisation application operating

Here the figure shows that the two required file have been loaded, on the top of the application there are the complete path. Then one of the gxl files has been selected and the application has directly drawn the corresponding image with the graph above. The nodes are colored by feature, it means that every type of node has its own color (e.g. lymphocyte nodes are red). By default the transparency is initialized to 125, which makes the image looks enough clear to correctly see the graph. When the user is satisfied of his configuration, he can decide to save the images one by one (it can be useful if he wants a different configuration for each image) or directly save all images with the same configuration.

5 Conclusion

The project was divided into two part quite different in their implementation but are still related because the both have a common goal, which is to provide tools that take a data set as input and convert it into another one. Also every application is related or directly work with the image data (hotspot and whole slide image).

The data processing part allows to convert one (or more) specific type of data into another format. We have used the different libraries offered by python in order to read and write the different type of data. This approach is very powerful and efficient, especially with the formats that are very known and used (e.g. json format because there is a lot of documentation about it and his corresponding python library). Although this is possible to find a library for almost each data, this is not the only way to process data. An alternative way could be to write manually every element of a specific format (e.g. for a xml file, with a library, it would automatically write the node '<element>...</element>' by simply calling a method. An alternative could have been to work only with the strings and write "manually" the xml file).

The visualisation part aims to draw graph on hotpsot images. For this purpose, the user had to give a gxl file that contains all the information about the graph. The goal was not only to draw graph but to implement an interactive interface. Here again there are many different possibilities to achieve it. For this project we used a python library (tkinter) that provide tools to create each component onto a grid. As we have seen in the result (Figure 13) there are already many options to customize the graph but this is opened to add more according to the wishes of the user.

With this project we have worked with a plenty of different data type and in order to deal with them the main strategy was to use some libraries. We also see how informatics can help medicine stuff with these different applications, although there are a lot of other ways.

References

[1] L. Studer, "BTS-Project A Combined Budding/T-Cell Score (BTS) in pT1 and Stage II Colorectal CancerPhD Thesis Linda Studer." `https://icosys.ch/bts-project`. Accessed: 2022-10-10.

[2] H. Dawson, L. Christe, M. Eichmann, S. Reinhard, I. Zlobec, A. Blank, and A. Lugli, "Tumour budding/t cell infiltrates in colorectal cancer: proposal of a novel combined score," *Histopathology*, vol. 76, no. 4, pp. 572–580, 2020.

[3] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta, "Comparison of json and xml data interchange formats: a case study.," *Caine*, vol. 9, pp. 157–162, 2009.

[4] J. A. Bondy, U. S. R. Murty, *et al.*, *Graph theory with applications*, vol. 290. Macmillan London, 1976.

[5] G. Jaume, P. Pati, V. Anklin, A. Foncubierta, and M. Gabrani, "Histocartography: A toolkit for graph analytics in digital pathology," in *MICCAI Workshop on Computational Pathology*, pp. 117–128, 2021.

[6] P. Pati, G. Jaume, A. Foncubierta, F. Feroce, A. M. Anniciello, G. Scognamiglio, N. Brancati, M. Fiche, E. Dubruc, D. Riccio, M. D. Bonito, G. D. Pietro, G. Botti, J.-P. Thiran, M. Frucci, O. Goksel, and M. Gabrani, "Hierarchical graph representations for digital pathology," in *Medical Image Analysis (MedIA)*, vol. 75, p. 102264, 2021.

[7] N. Farahani, A. V. Parwani, L. Pantanowitz, *et al.*, "Whole slide imaging in pathology: advantages, limitations, and emerging perspectives," *Pathol Lab Med Int*, vol. 7, no. 23-33, p. 4321, 2015.

[8] P. Bankhead, M. B. Loughrey, J. A. Fernández, Y. Dombrowski, D. G. McArt, P. D. Dunne, S. McQuaid, R. T. Gray, L. J. Murray, H. G. Coleman, *et al.*, "Qupath: Open source software for digital pathology image analysis," *Scientific reports*, vol. 7, no. 1, pp. 1–7, 2017.

[9] N. M. Jawhar, "Tissue microarray: a rapidly evolving diagnostic and research tool," *Annals of Saudi medicine*, vol. 29, no. 2, pp. 123–127, 2009.

[10] C. Chapman, P. Wang, and K. T. Stolee, "Exploring regular expression comprehension," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 405–416, IEEE, 2017.

[11] C. Chapman and K. T. Stolee, "Exploring regular expression usage and context in python," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 282–293, 2016.

[12] M. Hachicha and J. Darmont, "A survey of xml tree patterns," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 1, pp. 29–46, 2013.

[13] R. C. Holt, A. Schürr, S. E. Sim, and A. Winter, "Gxl: A graph-based standard exchange format for reengineering," *Science of Computer Programming*, vol. 60, no. 2, pp. 149–170, 2006.