UNI
FR
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

Supervisors:
Linda Studer
Rolf Ingold

University of Fribourg
DIVA research group
1700 Fribourg
Switzerland

December 2022

Abstract

The detection of tumour and t-cell is a very important topic in digital pathology/colorectal cancer. It helps doctors to decide which treatments their patients have to follow in order to fight against 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 converting some data in the form of annotations into another data type that can be used in different software to visualize tumour buds and t-cell. The main goal is to make data more readable and usable for data analysis such as Machine Learning/Deep Learning. Visualisation is achieved by drawing of graphs on hotspots images showing a specific area containing tumour buds 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, digital pathology, data processing, graph visualisation.

Contents

1 Introduction

1.1 Generality

This thesis is talking about some applications which are useful for digital pathology purposes. Especially the detection of tumour buddings and T-cells in order to identify risk factors for colorectal cancer (CRC). It helps doctors to decide which treatment has to be taken for CRC. CRC is a important topic, as according to swiss cancer screening [1], the colorectal cancer is one of the most common cancer (the third one in Switzerland) with more than four thousand new cases which are registered every year. According to the papers [2, 3] tumor buddings are the appearance of lone tumor cells or tiny tumor cell clusters which are dissociated from the primary tumor mass and are located in the stroma (= "cells and tissues that support and give structure to organs, glands, or other tissues in the body" [4]). Pathologists use these cells as a prognostic factor. The detection of tumor budding give additional information about which treatment choose depending of the cancer stage. For example, in the early stage of CRC, it plays a role in the decision of making colorectal surgery or not. It can also help to identify the presence of nodal metastasis [5] which is lymph nodes that have developed a cancerous infection. On the other side, T-cells are a type of white blood cell, they assist in preventing infections and might aid in the battle against cancer [6]. The article [7] demonstrates that T cell combined with tumor is a prognostic factor as well. Particularly by computing for each interesting areas the budding/T cell scores (BTS), which is the number of tumour buds divided by the number of T cells. The results show that using tumor buds and T cells data together has increase the chances of survival. After several examinations, pathologists construct a pathology report [8] which gives a description of the state of cancer and a diagnostic in order to decide for the appropriate treatment.

For this project, a plenty of data is available in image format coming from tiny slides of colon tissue which have been stored in paraffin blocks and digitized. From these ones, another type of data is extracted as a textual format that contains annotations and also other features. The aim of this project is to use these data (images and annotations) for some applications. This project 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, where two of them are about passing from JSON to XML file [9] and the last one is from JSON to CSV file. The primary goal of that is to make data compatible with different frameworks as Linda's data processing pipeline [10].

The second part is about graph visualisation, where the basic idea is to draw a graph [11] on a hotspot image. The goal is to represent the structure of the different cells in an abstract way using graphs. Graph theory is a very large topic of mathematics; however, only the basics notions are needed for this project. Mathematically, a graph $G = (V, E)$ is a couple of two sets, V is the set of nodes and E the set of edges. Two nodes can be connected by an edge but it is not an obligation (i.e. a node can stand alone). A graph can be directed, i.e. the edges have a specific direction (one node is the input and the other one is the output). For this task, the graph dataset only contains undirected graphs (see Figure 1 as an example).

Schematically, the representation of a hotspot image by a graph is shown in Figure 2.

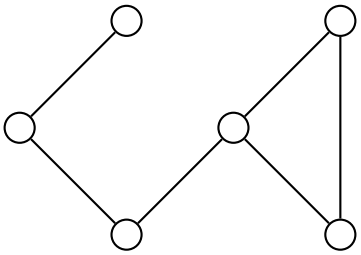


Figure 1: Example of undirected graph

Every cell (tumor bud and lymphocyte) are represented by the nodes and distinguished by a color (e.g. red for tumor buds and green for lymphocytes). Meanwhile edges represent the structure of the hotspot and the relationship between cells.

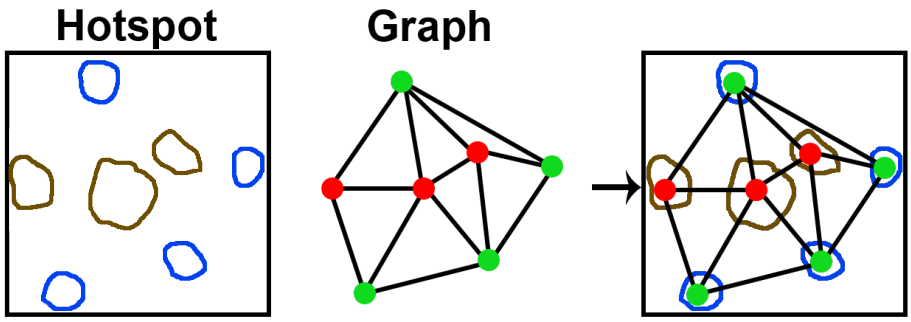


Figure 2: Schema of a graph on a hotspot image

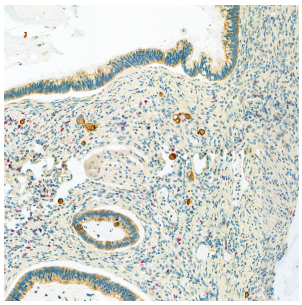
This paper is related to the PhD Thesis of Linda Studer [12]. It is structured as follow: First of all, the section 2 about technologies explain which tools are used in order to solve the problem. Then, the data processing (section 3) and the visualisation part (section 4) are described in the same way. An illustration of the strategy to suit and an overview of the implementation. In the end of each section comes a short description of some results.

1.2 Problem statement

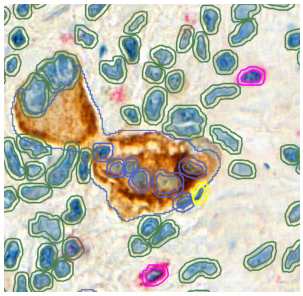
Initially the data comes out of pathology laboratory, where they are firstly processed on the software called **QuPath** [1]. It is an open source bioimage analysis software [13] which enables user to load and analyze whole slide image (Figure 4) or particular hotspot image (Figure 3). Data are represented as objects, i.e. each region of interest (e.g. a tumour bud) is an object

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

that can be handled. A functionality of QuPath is to highlight the regions of interest by using annotations (e.g. polygon, circle, rectangle, etc...). For example, the brown spot in the Figure 3b is a tumour bud which is bounded by a blue polygon. Another feature of QuPath is that objects can be classify and have different types in order to speed up the search time of an object. Further the structure of the objects is in the form of a hierarchy which also helps to decrease the search time.



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



(b) Hotspot image open on QuPath

Figure 3: Hotspot images (png format)

A benefit of QuPath is that each annotation firstly contains the coordinates of the polygon but can also have extra features which are stored as key-value database. Furthermore these ones are extracted and saved into text files using *JSON* format. *JSON* is a very efficient and convenient format to store data due to the fact that it is easy to read and use. A lot of programming languages implement libraries which give the capacity to process this type of file (e.g. Python, Java). The full list of the features is the following:

- 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).

The data processing part of this project is divided into three different conversions. The first one is about to convert the ROI points of each annotation of an hotspot image into a file readable by the software **ASAP** [2]. Subsequently, by using this file, the annotations can be drawn on the whole slide image [14] (Figure 4). This one is obtained from a scanner that digitize glass slides which save the image into an *mrxs* file. This is a large type of data since

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

its size is between 90’000 and 200’000 pixels. Although QuPath is also able to load *mrxs* files, the inconvenient is that it is much slower to process whole slide image than ASAP.

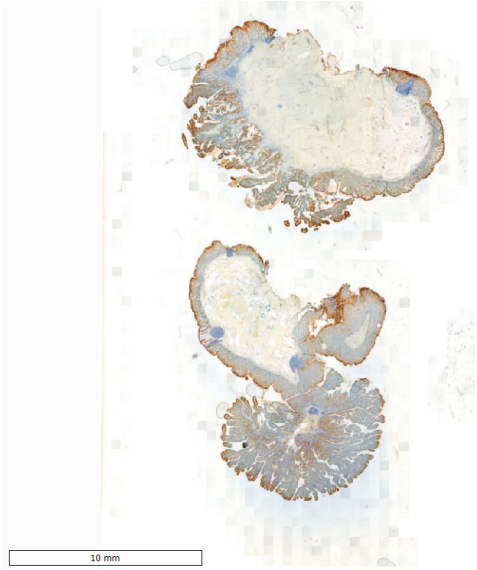


Figure 4: Whole slide image

For ASAP to be able to read and draw the annotations, it is required that the given file is of *XML* type with a specific structure. *XML* is a textual data format which stores data as a tree structure [15]. A tree is a data structure compose of a set of node, a root node and edges between the nodes. Each node (or element) has a predecessor (or parent) except the root one and one or more successors (or children/subelements) except the leaf nodes (the last nodes of the tree). Every element is built with a name tag and can contain either attributes or values to store data. This type of model is very efficient especially for searching for a particular information. *XPath* [16] and *XQuery* [17] are particularly two languages which have been developed for this purpose.

In order to make the XML tree readable by **ASAP**, it must have the following structure:

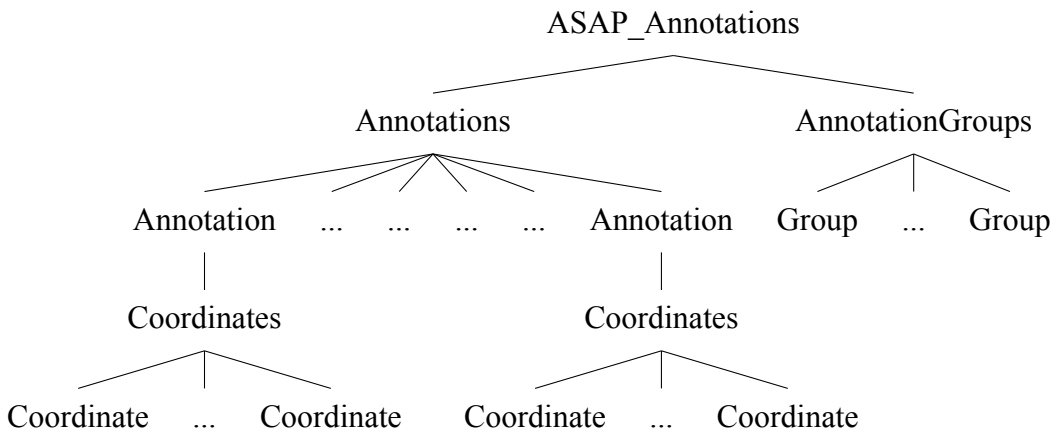


Figure 5: XML tree readable by ASAP

The root node is **ASAP_Annotations** and 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 indicates 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 XML code below (Source Code 1) is an example of file following the structure of the Figure 5.

```
<ASAP_Annotations>
    <Annotations>
        <Annotation Name="Annotation 0" PartOfGroup="Tumor" Color
   ="#4d66cc" Type="Polygon">
            <Coordinates>
                <Coordinate Order="0" X="24" Y="72"/>
                <Coordinate Order="1" X="34" Y="65"/>
                ...
            </Coordinates>
        </Annotation>
        ...
    </Annotations>
    <AnnotationGroups>
        <Group Name="Tumor" PartOfGroup="None" Color="#4d66cc">
            <Attributes/>
        </Group>
        ...
    </AnnotationGroups>
</ASAP_Annotations>
```

Source Code 1: XML file example containing annotations

Each **Annotation** element contains an identifier (name), a group (type of cell), a color and a type (shape of the annotation) written as XML attributes. The coordinates of the annotations are also stored as attribute in the **Coordinate** element where the *Order* attribute is for indicate in which order the polygon has to be drawn (i.e. for n points, ASAP draws line from 0 to 1, 1 to 2, ..., n to 0) and the *X* and *Y* attributes are for the standard coordinates of a point.

A problem is that the JSON file only contains the coordinates of the annotations in the hotspot image (Figure 3) and not in the whole slide image (Figure 4).

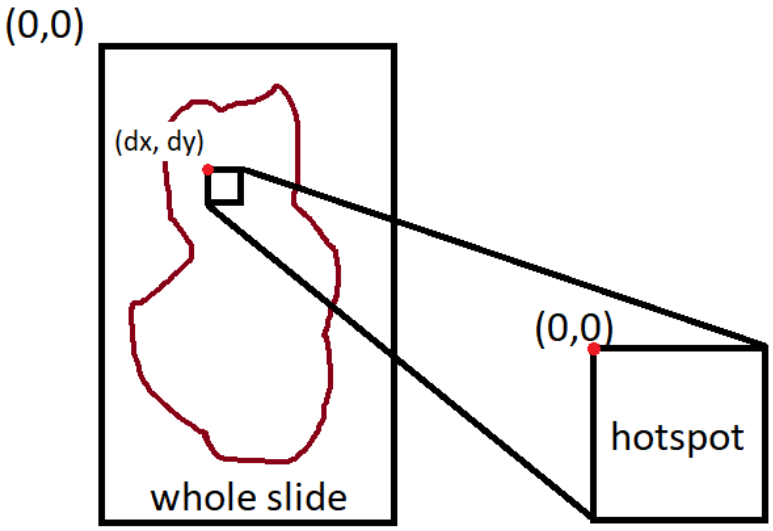


Figure 6: Schema of the hotspot position on the whole slide image

Figure 6 illustrates how the coordinates has to be shifted correctly in order to draw the annotations in the right place. To do so, for every JSON file there is an associated XML file holding the coordinates of the square where the hotspot has to be on the whole slide image (especially the (dx, dy) as shown in Figure 6). This particular file can be represented by the structure displayed in Figure 7.

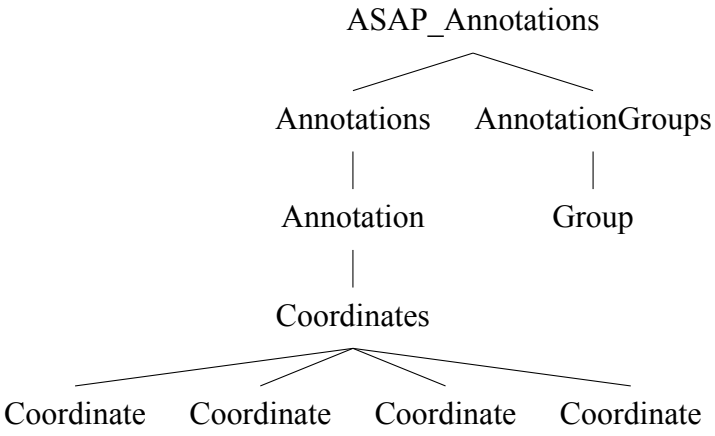


Figure 7: xml tree for the hotspot coordinates

The structure is very similar to the one of the Figure 5 but this time the XML tree only contains coordinates of the hotspot (four **Coordinate** elements because the shape is a square) and has only one group (hotspot group).

The second conversion works with image displayed in Figure 8. It is another type of image which is named tissue micro array (abbrev. TMA). TMA can be seen as a combination of hotspot (3) and whole slide image (Figure 4). As the whole slide image, it is an *mrxs* file; hence, it can be open onto ASAP as well. According to the paper [18], the process to produce a micro tissue array is firstly to cut donor blocks (i.e. blocks that contain tissue which eventually owns cancer cells) into many microscopic slides. Then the areas of interest are marked and finally transform into an array arranged in a paraffin block.

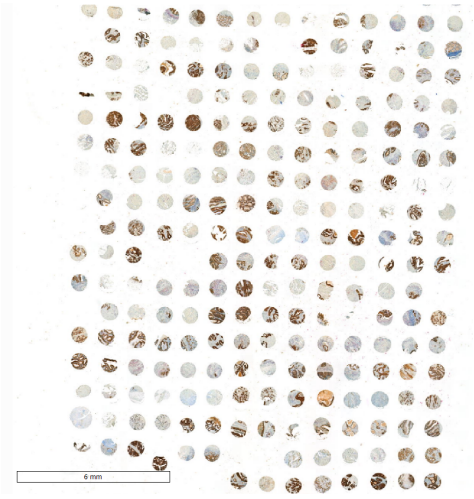


Figure 8: TMA

Compare to hotspot images (Figure 3), TMA is basically an array of hotspots but each sample has a circular shape instead of a square one and is smaller in size. The main advantage

of TMA over single hotspot image is that it holds a lot of different areas of interest rather than just one.

The goal of this conversion is the same as the first one which consist to convert the JSON file of each hotspot image into an XML file. The file has also the same structure (5) since the tissue micro array is also processed in ASAP. Hence, the same problem of coordinates shown in Figure 6 occurs as well. In order to deal with it, a CSV file has been processed. CSV (comma-separated values [19]) is a format that stores data as a two-dimensional array in a text file where the values in each row are generally separated by comma (or other delimiters as semi-colon for example). This type of file can be open on Microsoft Excel [3] in order to process the data. Figure 9 shows the content of a sample CSV file, it holds the coordinates of the center of each circle of the TMA, the radius and IDs.

	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 9: CSV containing hotspot coordinates

Finally, the last conversion focus on the other one-dimensional features of the JSON file that have not been used yet (e.g. Area, Circularity, Classification, Solidity, ...). The purpose is to store these features into a CSV a file in order to be able to visualise and process data on Excel.

The second part of the project is about the visualisation which consist to display a graph on an image. The goal is to provide a dynamical interface where the user can directly interact with it. The main part is to enable user to load (hotspot) images and another folder containing files with graph coordinates. These other files are of *GXL* (Graph eXchange Language) type [20] and for each hotspot images, they are generated through the Linda's data processing pipeline [10] with the help of the files obtained in the data processing part. GXL is a similar format to XML but is most suitable for storing graph coordinates. In the same way that XML proceeds, every elements are defined by tags, e.g. nodes (resp. edges) are represented by <node> (resp. <edge>). Thus, the structure of the *GXL* file can also be represented as a tree (Figure 10).

[3] Wikipedia: `https://en.wikipedia.org/wiki/Microsoft_Excel`

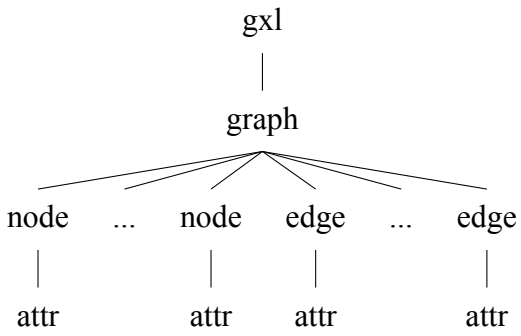


Figure 10: gxl tree

As shown in the Source Code 2 below, every node element has an ID and every edge contains the ID of two nodes in order to be able to properly draw a segment between two nodes. The attribute leaves (attr) of the nodes contain the coordinates (x,y) which is the position of the node on the image and can also own extra attributes as the type of a cell (e.g. 'tumorbud'). For the edges, the attribute leaves hold the distance between the both nodes composing an edge. The <graph> element is useful to indicate if the graph is directed or undirected as well as for saving the identifier (id).

```
<gxl>
  <graph id="00.5771_3D_AE1_AE3_CD8" edgeids="false" edgemode="
   undirected">
    <node id="_0">
      <attr name="type">
        <string>lymphocyte</string>
      </attr>
      <attr name="x">
        <float>83</float>
      </attr>
      <attr name="y">
        <float>102</float>
      </attr>
    </node>
    ...
    <edge from="_24" to="_26">
      <attr name="distance">
        <float>193</float>
      </attr>
    </edge>
    ...
  </graph
</gxl>
```

Source Code 2: GXL file example containing graph data

2 Technologies

There are a plenty of way to implement the different applications. For this project, the programming language Python [4] (version 3.9) is used. 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.). Anaconda environment [21] is associated to Python in order to create virtual environments. These ones are defined by a directory which holds all the installed packages and their versions (including python version). The main advantage to use virtual environments is that packages and python can have different version depending of which environment is used. Another benefit is that the code can be shared easily to someone because the person can automatically build the environment, i.e. owns the whole dependencies with the appropriate versions. The main packages that are used for this project are:

- **json** [22] (included with Python 3.9): The default library for processing JSON file containing the primary operations as opening, reading and writing.
- **lxml.etree** [23] (version 4.8.0) and **xml.dom.minidom** [24] (included with Python 3.9): Two different libraries which are used to process XML file. The first one is efficient for creating/writing XML tree; meanwhile, the second one is more convenient to open and read an XML file
- **csv** [25] (included with Python 3.9): The default library for processing CSV file containing the primary operations as opening, reading and writing.
- **argparse** [26] (included with Python 3.9): The efficient library for creating an argument parser, which enable the user to pass some parameters through a command line.
- **os** [27] (included with Python 3.9): The core library for manipulating paths of directories and files in the operating system.
- **re** [28] (included with Python 3.9): The library for construct and use regular expressions. Particularly efficient when a given string has to match with a specific format.
- **OpenCv** [29] (version 4.5.4): A multitask library for processing image; for instance, it provides some tools to draw shapes on an image (e.g. circle and line for drawing graphs).
- **matplotlib** [30] (version 3.4.3): A very large and popular library which gives tools for visualisation.
- **numpy** [31] (version 1.16.6): The library needed for doing advanced mathematical operations and constructing multidimensional arrays.

[4] Website: `https://www.python.org/`

- **tkinter** [32] (version 8.6.11): A convenient library to create a graphical user interface (GUI) in Python.

In order to generate the appendix (section 5) of this paper, which contains the documentation of the python projects for the both part (i.e. data processing and visualisation), the documentation generation **Sphinx** is employed. It enables to construct a documentation from a python project in several formats, as LaTeX, HTML, ...

3 Data Processing

3.1 Strategy

Data processing consist of making some conversions, mainly from the JSON file described in problem statement to other format as XML and CSV. As mentioned before in the chapter 2, all programs are implemented in Python; hence, an efficient way to perform conversions between two data type is to make usage of libraries. There are three different conversions based on the different images available.

1. **JSON** (+ XML) $\rightarrow$ **XML**: The goal is to process the JSON file containing annotations of each hotspot image to transform it into an XML file that can be read by **ASAP** (i.e. the output file must have the structure illustrated in Figure 5), which can draw the hotspot on the whole slide image (Figure 4). In order to correctly shift the coordinates as described in Figure 6, other XML files holding hotspot coordinates are available (Figure 7).

   To begin with the strategy, the **json** library can be used to open and read JSON files; hence, it is possible to read the coordinates of the annotations. The next step is to add to these coordinates the coordinates of the top left corner of the hotspot square as shown in Figure 6. The top left corner coordinates can be extracted by reading the corresponding XML file with the **xml.dom.minidom** library. Mathematically, assume that (X_{ik}, Y_{ik}) is the k-th point of the i-th annotation and (dx, dy) to be the top left corner coordinates. Thus, the final coordinates to write into the output file are simply $(X_{ik} + dx, Y_{ik} + dy)$ with the help of the **lxml.etree** library which enables the construction of XML tree.

2. **JSON** (+ CSV) $\rightarrow$ **XML**: This second 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 which can be processed by **ASAP** for the tissue micro array (Figure 8). The difference is that the hotspots are no longer squares but circles and the file containing the data for the offset is a CSV file (as displayed in Figure 9) instead of XML.

   The strategy is to read the JSON in the same way than conversion 1 to get the coordinates (X_{ik}, Y_{ik}) as previously defined. The CSV file, which gives the coordinates of the center of the circle (Cx, Cy) and the radius r, is read with the **csv** library. As the hotspot is a circle, (Cx, Cy) has to be add to (X_{ik}, Y_{ik}) and subtract by the radius r in order to correctly shift the coordinates. Thus, the final coordinates to write into the output file are $(X_{ik} + Cx - r, Y_{ik} + Cy - r)$.

3. **JSON** $\rightarrow$ **CSV**: In the previous conversions, it only uses the ROI (region of interest) points of the JSON file. While this last conversion processes the other features (e.g. Area, Circularity, Classification, Solidity, ...) and stores them in a CSV file.

   This task can be directly solved by translate the data from JSON to CSV. As the previous conversions, JSON files are open by the **json** library. Each interested features are read and written into a CSV file output by using the **csv** library.

For the user to give the desired files as input, the **argparse** library perfectly fit this purpose. It enables to indicate the directory path where the files are located (or directly the file path) in a single command line. The **os** library is also crucial to navigate between the different file/directory paths. Furthermore, the files provided must have the appropriate format; hence, the **re** library is used to create regular expression in order to verify the validity of the file name and extension (e.g. .json, .xml, ...).

3.2 Implementation

3.2.1 Argument parser

Each conversion has to implement an argument parser that enables the user to pass the files through a command line. With the library **argparse**, a list of arguments can be added to enforce the user to fill them, otherwise the program inform that the conditions are not met and stop the execution. For example:

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

A new parser is created and the argument *input_file* is added with the attribute type set to string (str) and "required" set to true. It means that the user has to pass a string for the argument *input_file*. The variable **args** is necessary to parse the argument (i.e. to obtain the value given by the user). For example, the code can be launched by opening a command shell and typing the following command line:

python example.py –input_file exfile.json

The python script *example.py* is containing the argument parser and the file *exfile.json* is the JSON file given as input.

The second step is to check the files which have been passed in the command line. It is done by the usage of regular expression. For the three conversions, a directory containing JSON files has to be given as input; hence, a regular expression is needed for verifying in the folder which one have the JSON extensions and which do not. Additionally, it must not take into account the JSON files which do not have a valid name (e.g. a valid file containing data about hotspot could 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* for example, it must not be considered because the name is clearly not the accepted format). To do so, a regular expression [33, 34] can be implemented using the **re** library. It can be represented by the following skeleton:

$$Masks\_[A\text{-}Za\text{-}z0\text{-}9]^{*}\text{-}level0\text{-}hotspot.json$$

This regex means that the string must start with the string *Masks_*, followed by any characters (0 or more) and ends up by the string *-level0-hotsport.json*. It can be implemented by using the search or match method from the **re** library. For example, the search method takes as input a regex expression and a test string:

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

In the example above, the result of the search method is true as the file is validate by the regex (note that: **.*** means any characters 0 or more).

3.2.2 Open and read a JSON file

As explained in the strategy 3.1, all conversions take a JSON file as input; hence, the opening and reading are implemented in the same way. Firstly, the file is opened using the **open** method of Python and loaded with the **json** library:

```
with open(file_to_process) as file:
    # load json file
    data = json.load(file)
```

Afterwards, the data can be simply read as a Python dictionary, e.g. a simple JSON file could be:

```
{
    "id" : 3
    "name" : "Tumor"
}
```

The value of the key ”name” can be accessed by writing:

```
cell = data['name']
# cell = "Tumor"
```

3.2.3 Open and read XML file

In addition to opening a JSON file, conversion 1 has to open an XML file containing the coordinates of the hotspot square (7) in order to correctly shift the data on the whole slide image (Figure 4). To do so, the **parse** method of the **minidom** library is used to open and parse the XML file; moreover, the values of the elements **Coordinate** are extracted with the help of **getElementsByTagName** method. The attribute values holding the (X, Y) coordinates of the hotspot square can be easily obtained with the field *attributes* from the XML element and stored in a list:

```
# open xml file
hotspot_file = minidom.parse('coordinates.xml')
# read coordinates
coordinates = hotspot_file.getElementsByTagName('Coordinate')
hotspot_coord = [(float(point.attributes['X'].value),
                  float(point.attributes['Y'].value)) for point in
                  coordinates]
# extract top left corner coordinates
dx, dy = hotspot_coord[0]
```

At the end, only the coordinates of the top left corner (dx, dy) are required in order to correctly shift the coordinates according to the Figure 6.

3.2.4 Open and read CSV file

The last input file to open is the CSV file from the Figure 9, which gives the data for shifting the annotations of the hotspot on the tissue micro array (Figure 8). As JSON file, CSV is opened with the **open** method from Python. The **reader** method create an iterator that can be used to go through the CSV file row by row.

```
    # open csv file
    with open(coord_file) as csv_file:
        # iterator for reading the csv file
        reader = csv.reader(csv_file, delimiter=';')
```

As illustrated in the code below, the CSV file is read with the help of the iterator; hence, the remaining part is to find out the row corresponding to the hotspot that is being processed. It is done by comparing the ids of the JSON and CSV file. Thus, the coordinates of the center and the radius can be directly read as shown below:

```
    # reset the iterator to the begin of the file
    csv_file.seek(0)
    # iterate over every row
    for row in reader:
        # search for the line containing the id of the current hotspot
        if current_id == str(row[0]):
            # extract the center coordinates and radius
            coord = (float(row[1]), float(row[2]), float(row[3]))
    return coord
```

Note that every time the iterator go through the file, it must be reset to the begin with **seek** method; otherwise, the loop starts from the last line that was read and not from the beginning.

3.2.5 Write a XML tree (conversion 1 and 2)

For the conversions 1 and 2, the output is an XML file that can be read in **ASAP**. Then for the both conversions, the goal is to construct the XML tree displayed in Figure 5. The implementation of it with the help of **lxml.etree** (renamed **ET** in the code) works as follow: The nodes are defined from top to bottom (i.e. from the root node to leaf nodes). More precisely, it begins with the construction of the root node **ASAP_Annotations** with the **Element** method:

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

For the construction of the rest of the tree, the **SubElement** method, which takes the parent and name of the new node as parameter, is used to create a new node from a previous one. Then for the branches **Annotations** and **AnnotationGroups**, the implementation is the following:

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

Additionally, some nodes own attributes, as the **Coordinate** nodes which contain the coordinate of a ROI (region of interest) point (i.e. a point that builds up the polygon). The attribute has to be given as a Python dictionary to the **SubElement** method. For the both conversions (1 and 2), the ROI points are located in the JSON file. The offset, which is computed with the other input file (i.e. refer to section 3.2.3 resp. 3.2.4 for conversion 1 resp. 2), has to be added to each of these points. For the first conversion, the **Coordinate** are created as follow:

```
    # get the list of points
    points = object['ROI_Points']
    # iterate over the list of points
    for j, point in enumerate(points):
        # create the attribute
        coord_attrib = {'Order': str(j), 'X': str(point[0] + dx),
                        'Y': str(point[1] + dy)}
        # create Coordinate element
        xml_coordinate = ET.SubElement(xml_coordinates, 'Coordinate',
                                       attrib=coord_attrib)
```

The list of ROI points is obtained with the JSON file, so the attribute $'X'$ and $'Y'$ can be filled using the points and the offset. The $'Order'$ attribute indicates the number of the point, i.e. the order in which the points are linked. Notes: The values of an attribute have to be strings and not numbers. For the second conversion, the code is identical except the offset which is different as explained in the strategy part (3.1).

3.2.6 Writing a CSV file (conversion 3)

For the last conversion (3), the way for saving features into a CSV file is to create a dictionary for each row as follow:

```
    with open(output_file, 'w', newline='') as csv_file:
        # csv file define as a dictionary writer
        writer = csv.DictWriter(csv_file, delimiter=';',
                                fieldnames=features)
        # write the title of each column
        writer.writeheader()
        # write rows
        for obj in objects:
            row = {feature: obj[feature] for feature in features}
            writer.writerow(row)
```

Firstly, the output file is opened and initialize for writing with the **DictWriter** method. The selected **features** are saved as strings into a list and defined the name of each column. Finally, for each interested features of the JSON files, their values is extracted and written into a dictionary which is used to write a row with the **writerow** method.

3.3 Validation

For the first conversion (1), as explained before, the XML file generated by the script can be loaded into **ASAP**. Hence, the validity of the application can be easily checked by loading the output file to the corresponding whole slide image (Figure 4).

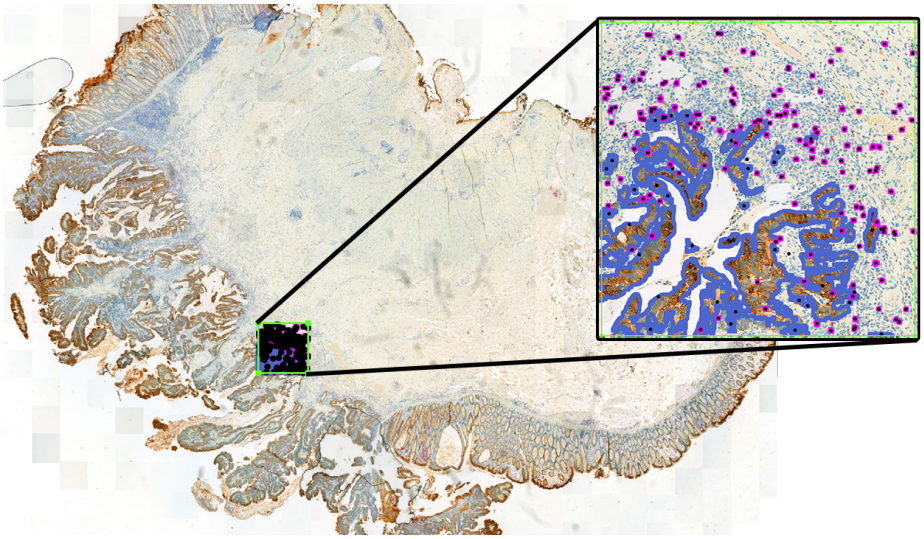


Figure 11: Whole slide image with the drawing of the hotspot

The second conversion (2) works on tissue micro array image (Figure 8) with the same principle as the first one. When loading the XML output file, **ASAP** automatically draws the annotations on the corresponding hotspot (Figure 12).

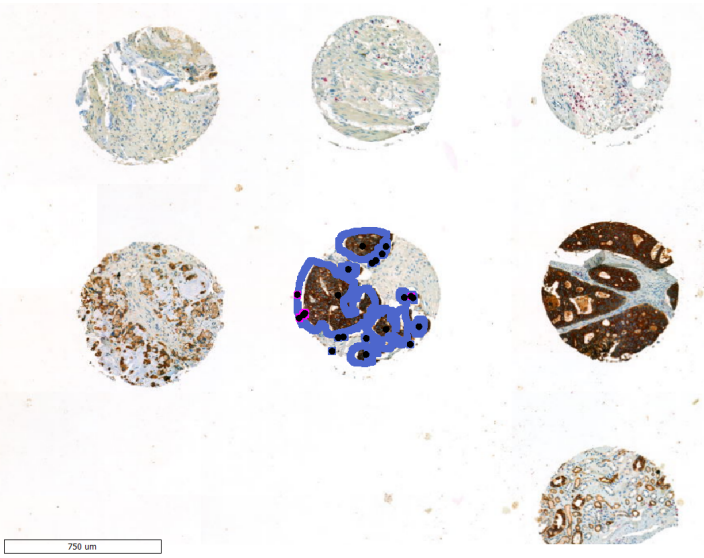


Figure 12: Drawing of the circular hotspot

For the last conversion (3), a CSV file is created with the additional features that relate to the annotations. In order to check for 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 13: csv output file loaded on excel

The table above is a very useful and efficient tool, as it gives an easy access for other analysis, e.g. for statistics (e.g. diagram, plot can be done on Excel) or in this case, as graph features.

4 Visualisation

4.1 Requirement

The visualisation part is about displaying a graph on an (hotspot) image. The goal is to provide a dynamical interface where the user can directly interact with it. The main part is to enable user to load an (hotspot) image and a GXL file containing graph coordinates (Figure 10).

4.1.1 Functional

The graphical user interface has to fulfill the following functional requirements:

- Load folder (image and GXL): The user can give the path to an image and GXL folder. In addition, he must have the possibility to select a specific GXL file and it would directly display the image with the graph.
- Draw graph: The core feature, the application has to draw a graph on an image with the help of a GXL file.
- Edit the color of nodes and edges: The user can decide for the color of the nodes and edges. In addition, they can specify nodes to be of the same color or to be colored by graph features (e.g. the nodes could be colored by name; for instance, red for ”tumor-bud” and green for ”lymphocyte” nodes).
- Edit the size of nodes and edges: The user can change the value of the radius of the nodes (which are represented by filled circles) and the thickness of the edges.
- Edit the transparency: The user can be able to change how the background image is visible.
- Edit the scaling: The user can modify the scaling of the graph (in case the coordinates are not in pixel).
- Save images: The user can choose for an output directory and the images can be either saved one by one or all at once.
- Searching for a specific GXL file: The list of GXL is often very large; hence, the user can search for a precise GXL file by typing is name (or a part of it).
- Select blank background or leave empty: When an image has not been found for a GXL file, the user can decide if the graph is still drawn on a blank background or leave the canvas empty.

These features must be dynamic, i.e. the user can see the current state of the image (graph + hotspot) in real time 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.

4.1.2 Non-functional

In addition to functional requirements, visualisation has also some non-functional requirements:

- Position of the components: For an adequate readability, the different components have to be placed in a consistent way. More specifically, the main component has to be the canvas on which the image is displayed with its corresponding graph. Therefore, it must be located in the middle of the screen and have a large size. All the widgets concerning the customization of the appearance of the graph has to be placed together (e.g. on the right side of the canvas).
- Permission to use a widget: The user may not be able to use some components if some prerequisites are not satisfied; for instance, the widgets for modifying the color of nodes have to be usable only when a graph is displayed.
- Input validation: The values given by the user has to be tested before performing an action; for instance, the thickness of the edges needs to be changed, string must not be accepted as a possible value but only numbers.
- Handle empty values: When a value is empty, e.g. the user erase the value for the size of the edge, the application has to rewrite the previous value. Also, whether no GXL file is selected, the previous file that has been used to draw the graph has to be reused.

4.2 Strategy

The core of this part is to construct a graphical user interface (GUI). A way to reach this goal is to use the **tkinter** library from Python as mentioned in section 2. The latter contains all the basic elements for building a GUI (e.g. button, label, canvas, scrollbar, ...). As illustrated in Figure 14, the operating principle of **tkinter** is to arrange the components into a grid. In the first place, a grid is initialized and the elements are arranged on it; in addition, the widgets can take more than one grid square. According to the non-functional requirement (4.1.2), the canvas is the main component; hence, it has to take the largest space on the grid.

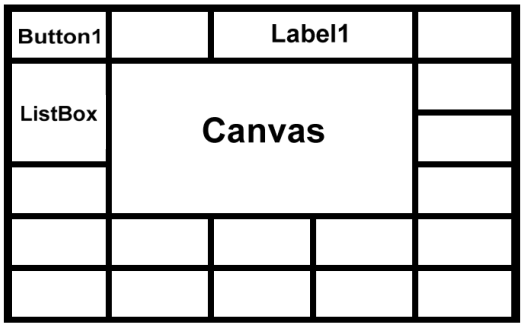


Figure 14: Example of a tkinter grid

Similar to the strategy of the data processing part (section 3.1), the visualisation part also needs libraries as **os** (for the processing of every file/folder path given in input). However it does not need to have the **argparse** library, since **tkinter** provide some methods to ask user for a directory path. The **re** library is also reused in order to find out the proper GXL file corresponding to the given image. As previously seen in the problem statement (section 1.2), GXL file are structured as XML file; hence, it can be parse using an XML library as **xml.etree**.

Once the files given by the user are correctly processed and parsed, the main part is to draw the graph on the appropriate image and displays it on the canvas. The strategy is to use the **OpenCV** library which enables to draw shapes on an image. The coordinates of each nodes and edges can be found in the content of the GXL file (as shown in Figure 10 and in the source code 2). Thus, when iterating over the GXL file, for each node (resp. edge) a circle (resp. a line) is drawn with the properties given by the user on the GUI.

The project can be represented by a class diagram as illustrated in Figure 15. The design pattern **Abstract Factory** is used in order to create the GUI efficiently. It is a creational pattern which provide an interface for building groups of related objects without defining their concrete class. In this case, it would offer the possibility to have a plenty of different GUIs. However, for this project, the GUI is only create with **tkinter** objects; hence, the pattern is not used to its full potential. Nevertheless, the code becomes more flexible for any future improvements.

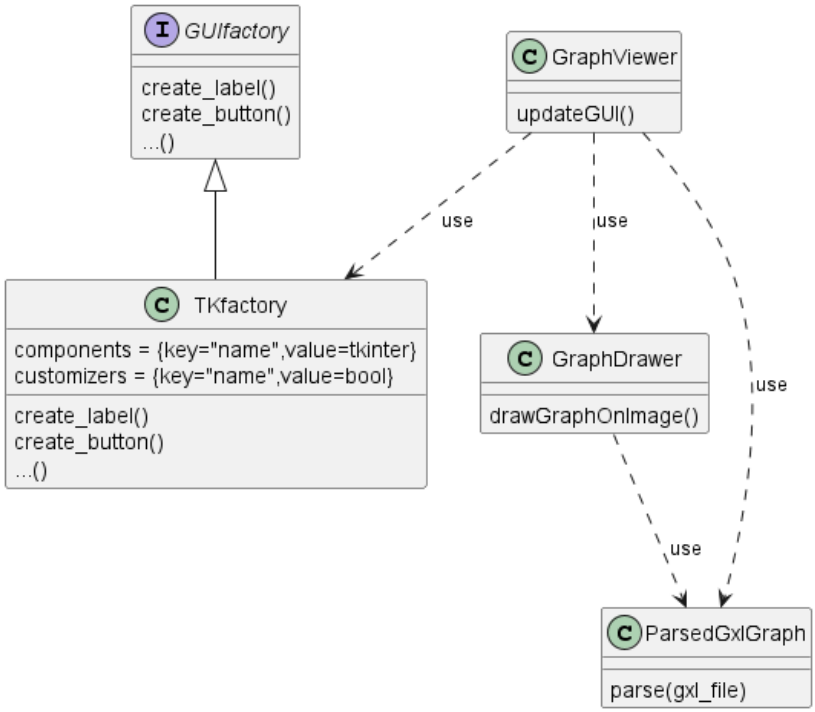


Figure 15: Class diagram for visualisation

- **GUIfactory**: The GUI interface, which provides all the methods to create a basic GUI.
- **TKfactory**: The concrete class which implements **GUIfactory** interface. It creates tkinter widgets and store them into a dictionary named *components*, which holds the name (string) as the key in order to identify the component and the tkinter object as value. In addition, the dictionary *customizers* store the name as the key and a boolean value which is true if the widget is a modifier (i.e. if the user interact with this component, it potentially changes the graph appearance).
- **ParsedGxlGraph**: This class parse a given GXL file in order to make it easier to access the different parts, as the node and edge features.
- **GraphDrawer**: This class draws a graph from a GXL file with the help of the **ParsedGxlGraph** class on an image.
- **GraphViewer**: The main class that keeps the GUI up to date and provides the methods related to the components. It uses **TKfactory** to get the components which have to be update as well as **GraphDrawer** in order to draw the image on the canvas. In addition, **ParsedGxlGraph** is used to fill the color by feature option menu (i.e. with which feature the nodes have to be colored).

4.3 Implementation

4.3.1 Creating the GUI

The first step of the implementation is to create the class **TKfactory** which is used to create the widgets using **tkinter**. All the components are implemented in the same way; for instance, a button is created as follow:

```
    def create_button(self, frame: tkinter.Frame, name: str,
                      is_customizer:  bool = False):
        button = ttk.Button(frame)
        self.add(name, button, is_customizer)
        return button
```

The button is created on a frame, which can be considered as a subset of the full grid (Figure 14). Each component is identified by a unique name. The *add* method stores the button in two dictionaries: The first one containing the name as key and the tkinter component as value is useful to access and modify the object. The second dictionary stores the name and a Boolean value which is true if the component is a modifier, i.e. it can interact with the graph image. This structure is important for managing user permissions to access a widget, as previously seen in section 4.1.2.

The position of the button can be specified after the creation by calling *grid* method. For example, the following button is located at the top left corner of the grid and takes two columns of space:

```
    myButton = tk_factory.create_button(mainframe, 'myButton')
    myButton.grid(column=0, row=0, columnspan=2)
    myButton.configure(text='myTitle', command=myMethod)
```

In addition, a configuration can be added to the button in order to assign it some properties. The main properties are the title (i.e. the name that is displayed on the GUI) and the command (i.e. the method that is executed when pressing the button).

4.3.2 Open files

The user has to be able to pass an image and GXL files folder. In addition, the GXL file has to be matched with its corresponding image. Firstly, two buttons are created, one is dedicated for the image folder path and the other one for the GXL folder path. A method is associated to each button which uses the *filedialog* object of tkinter to ask the user for a directory:

```
    def select_img_dir(self, *args):
        # ask user for a directory path
        new_dir = filedialog.askdirectory()
        if new_dir:
            if os.path.isdir(new_dir):
                # if the directory path is valid, update the variable
                self.img_dir.set(new_dir)
```

This method is a part of the class **GraphViewer** as it is related to a tkinter widget (button). After checking if the path was valid, it is saved into a tkinter variable.

For the matching of GXL file and image, the usage of regular expression is again necessary. Firstly, the GXL files has to be displayed into a list box where the user can select one file and the application can draw the corresponding image on the canvas. The list box on tkinter works as follow:

```
    for file in sorted(os.listdir(new_dir)):
        if file.lower().endswith('.gxl'):
            if os.path.isfile(os.path.join(new_dir, file)):
                listbox.insert(END, file.rsplit('.', 1)[0])

    # get the selected file
    index = int(listbox.curselection()[0])
    gxl_filename = listbox.get(index)
```

The first step is to iterate over the directory and looks for file having the GXL extension. The extension is removed and the file is added at the end of the list. Thus, the user can select a file by clicking on it and the list box can detect which file has been selected with the method *curselection()* method to retrieve the index.

In order to find the corresponding image of a GXL file, the name of the GXL file is extracted by removing its extension (.gxl) as shown in the code above. Then, the remaining part is to iterate over the image folder and check for the name of an image to contain it. Formally, the regular expression is constructed as follow:

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

The match of this expression with a given string simply means that the concerned string contains the sub string *gxlfilename*.

4.3.3 Parse a GXL file

This part consist to read a GXL file (Figure 10) and extract the information contained inside. The file can be opened by using **lxml.etree** library and can be traverse from root to children. A child is access with *iter* method; for example:

```
    # open file
    tree = ET.parse(gxlfilepath)
    # get root
    root = tree.getroot()
    # access the children 'node' of the root
    nodes = root.iter('node')
```

After the opening of the file, the first step is to get the feature names; for instance, the nodes always hold the x and y coordinates as feature (refer to Source Code 2) and optionally other features as the type. In the code below, the feature names are extracted by going through the elements of the *mode* (= either node or edge) branch.

```
    features_info = [[feature for feature in graph_element] for
                     graph_element in root.iter(mode)]
    if len(features_info) > 0:
        feature_names = [i.attrib['name'] for i in features_info[0]]
    else:
        feature_names = []
```

In order to get the values of each feature, it is required to decode them as the type of value is denoted as a GXL element (e.g. *<int>34</int>*). It is simply done by reading the tag of the element and assign the corresponding type to the value.

```
    features = [[self.decode_feature(value) for feature in graph_element
                for value in feature if
                feature.attrib['name'] in feature_names]
                for graph_element in root.iter(mode)]
```

Thus, the code above produce a list of lists containing the values of each node for each feature, e.g. if the features names for nodes are the $type$, x and y then a resulting list of feature values could be: *[['tumorbud',13,14], ['lymphocyte',45,32], ...]*.

The last crucial information to extract is the edge list. Each edge holds the ids of the two nodes which compose the edge, e.g. *<edge from="_3" to="_7">*. The code below create two lists containing the starting and ending id. In order to extract 3 from "_3", the regex "_(\d+)" which accepts words that start with a "_" followed by one or more digits. Thus, by using *search* and *group(1)* from **re** library the regex can be matched with a given string and extracted as shown below:

```
    regex = r'_(\d+)$'
    start_points = [int(re.search(regex, edge.attrib['from']).group(1))
                    for edge in root.iter('edge')]
    end_points = [int(re.search(regex, edge.attrib['to']).group(1))
                  for edge in root.iter('edge')]
```

Finally, the edge list is produced by merging the two lists above and if necessary the values has to be shifted in order to match the index of node features and not their ids in order to directly access the node features list.

```
    if shift != 0:
        start_points = [x - shift for x in start_points]
        end_points = [x - shift for x in end_points]
        edge_list = [[start_points[i], end_points[i]] for i in
                     range(len(start_points))]
```

4.3.4 Customization options

In order to properly draw the graph, the user has to set a configuration (or use the default one), i.e. the size and color of the nodes and edges, transparency, etc. For the choice of a color, two clickable labels (node and edge) are used and displayed the chosen color as their backgrounds. In order to ask the user for a color, **tkinter** provides an efficient tool which is the *colorchooser* method. It can ask the user for a color by displaying this interface:

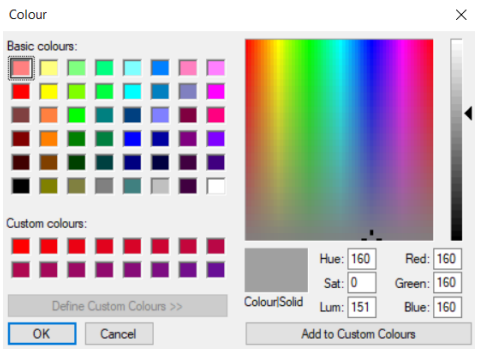


Figure 16: colorchooser

When the user select a color, the returned value is a tuple containing the RGB and HEX code format. In order to correctly update the color, the program still needs to know if the user has decided to color the nodes by feature or not (e.g. the nodes can be colored according their type or simply have the same color). This is the reason why the class **GraphViewer** has to access the parsed GXL graph in order to display in an option menu the different features available other than x and y which are reserved for coordinates. The option menu is implemented as follow:

```
    parsed = ParsedGxlGraph(os.path.join(self.gxl_dir.get(),
                            os.listdir(self.gxl_dir.get())[0]))
        features = [f for f in parsed.node_feature_names
                    if f not in ['x', 'y']]
        for feature in features:
            cbf_menu = self.components['cbf_menu']
            cbf_menu['menu'].add_command(label=feature, command=lambda
                            value=feature: self.color_by_feature.set(
    value))
```

The feature names of the nodes are stored in the option menu and each time the user change the selected feature it changes the *color_by_feature* variable. With **tkinter**, it is possible to trace a variable as shown in the code below. It means that a method can be called whether the value of the variable is changed.

```
    color_by_feature.trace('w', graph_viewer.onselect)
```

Note that in the code above, the method *onselect* from **GraphViewer** draws the graph on the canvas. It is called every time the value of *color_by_feature* is modified. This procedure is also called in all other cases where a value, which can affect the graph appearance, has changed. It uses the class **GraphDrawer** in order to draw the graph, as described in the section 4.3.5.

For the user to choose the size of node (radius) and edge (thickness), it is simply done by adding text fields where user can type the desired value. However, the application must not accept strings as input or empty content (according to the non-functional requirements). To do so, a short function is declared:

```
    def is_float(inp):
        if inp == '':
            return True
        try:
            float(inp)
            return True
        except ValueError:
            return False
```

This function returns true if the input is a number (float) and at first it authorizes the empty value. It can be linked to a text entry in order to control the values entered by the user as shown below:

```
    reg_is_float = frame.register(is_float)
    myentry = tk_factory.create_entry(frame, 'myentry')
    myentry.configure(validate='key', validatecommand=(reg_is_float, '%P
    '))
```

The function is primarily registered in the frame and then associated to the text entry. As the parameter *validate* is set on $'key'$, the function is executed when a key is pressed and if the result leads to a false then the key is not written at all. If the content of the entry is erased, the previous value has to be rewritten. It is done by reading the value in the appropriate variable; for example, if the thickness of the edge has been erased:

```
    if es_entry.get() == '':
        # if user pass no number for thickness -> rewrite previous one
        es_entry.insert(0, str(edge_style['thickness']))
    else:
        # update value
        edge_style['thickness'] = int(es_entry.get())
```

The two last element of configuration required before the graph can be drawn are the scaling and transparency. The scaling is a numerical value that the user enter to indicate the factor by which the coordinates are multiplied in case the scale is not in pixel; hence, a simple text entry is used. In addition, a key can be bind to a text entry in order to execute a method every time the key is pressed when using the entry. In the code below, the key "<Return>" has been bound. Thus, the *onselect* method is executed by pressing the return key.

```
    scaling_entry = tk_factory.create_entry(frame, 'scaling_entry')
    scaling_entry.bind('<Return>', graph_viewer.onselect)
```

The transparency of the background image is obtained by working with 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. Instead of create another text entry to write the transparency value, a practical way is to create a slider widget (*Scale* object on tkinter) with value between 0 and 255.

```
    transparency_scale = tk_factory.create_scale(frame, 'transparency')
    transparency_scale.configure(from_=0, to=255, orient=HORIZONTAL,
                                 length=255,
                                 command=graph_viewer.onselect)
```

Similar to button, a command attribute can be set which called a method each time the value is changed.

4.3.5 Drawing graph

Once all the configuration elements are set and the GXL file has been successfully parsed, the graph can be drawn 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 the **GraphDrawer** class where the image file, GXL file and the configuration are given. With the help of the GXL file parser class (4.3.3), the x and y coordinates of every node can be retrieved as well as the list of edges. Firstly, the points (coordinates of the nodes) are stored into a dictionary:

```
1    points = {i: tuple([int(x_y[0] * self.scaling), int(x_y[1] * self.
     scaling)]) for i, x_y in enumerate(self.graph.node_positions)}
2    # self.graph = Parsed GXL graph
```

Then the drawing of the edges works as follow:

```
1    for edge in self.graph.edges:
2        pt1, pt2 = (points[edge[0]], points[edge[1]])
3        img = cv2.line(img, pt1, pt2, color=self.edge_style['color'],
4                       thickness=self.edge_style['thickness'],
5                       lineType=self.edge_style['lineType'])
```

As seen in section 4.3.3, the edge list of the parsed GXL graph contains the index of the two nodes composing the edges. Hence, the coordinates are easily extracted from *points* dictionary previously defined. At the beginning, the variable *img* is containing only the image without any drawing on it; hence, after drawing all the edges, *img* contains the edges and the initial image as background.

The drawing of the nodes is similar to the edges except that circles are drawn for each point instead of lines. In addition, the color of the nodes can be different if the user choose to color them by the feature ”type”:

```
1     for feature, (i, pt) in zip(self.graph.color_by_features, points.
     items()):
2        img = cv2.circle(img, pt,
3                         radius=self.node_style[feature]['radius'],
4                         color=self.node_style[feature]['color'],
5                         thickness=self.node_style['thickness'])
```

The field *color_by_features* holds the list of the values of the given feature, e.g. for the feature ”type”, the length of the list is the same as the numbers of node (as ”type” is a feature of the nodes) and contain the values ”tumorbud” or ”lymphocyte”.Thus, the circle are drawn according to the node style of ”tumorbud” and ”lymphocyte”.

Afterwards the result image can be displayed on the canvas of the GUI and saved in an output folder. In addition, if no image correspond to a given GXL file, it is possible to draw the graph on a blank background using **numpy** as follow:

```
1    img = np.zeros([1024, 1024, 1], dtype=np.uint8)
2    img.fill(255)
```

4.4 Validation

The application works as follow: Before the user can interact with the different customisation features as modify color, transparency, etc. they 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 it, the program automatically draws 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, except if the check button for accepting blank background is checked (i.e. if no image is found, it still draw the graph on a blank background). Afterwards, it is possible to use the customisation panel on the right of the canvas. In practical, the application appears:

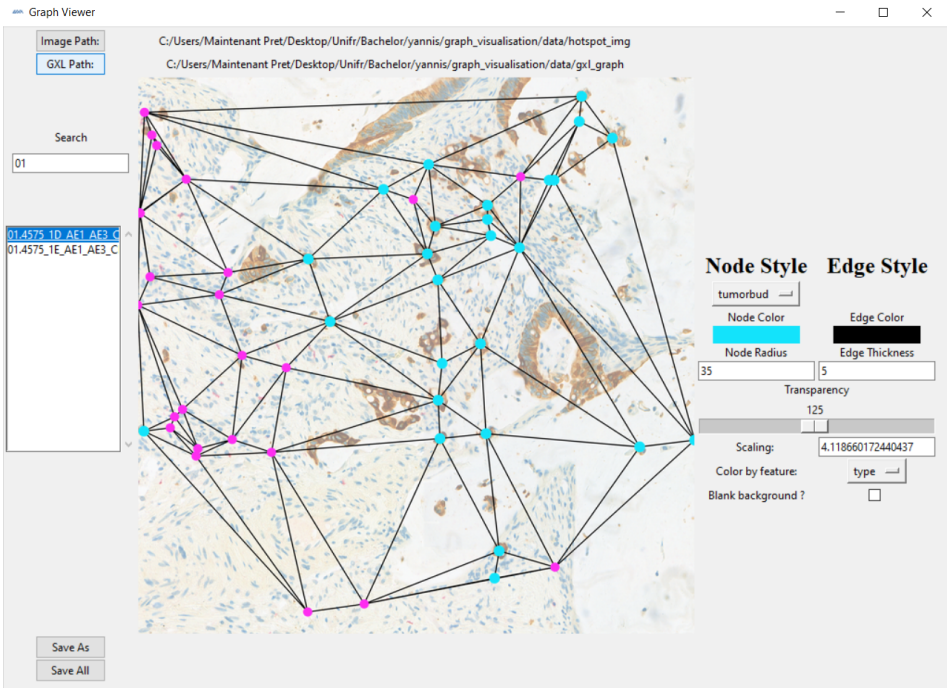


Figure 17: Graph visualisation application operating

Figure 17 shows that the two required files have been loaded, as their complete paths are displayed on the top of the application. 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 and tumorbud are blue). By default, the transparency is initialized to 125, which makes the image looks enough clear to correctly see the graph above. When the user is satisfied of his configuration, he can decide to save the images one by one (it can be useful if he desires a different configuration for each image) or directly save all images at once with the same configuration.

5 Conclusion

This project has shown a concrete example on how computer scientist can help digital pathologists with these different applications. It was divided into two parts quite different in their implementations 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 enables to convert one (or more) specific type of data into another format. Different libraries offered by python have been used 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 since a lot of documentation is available and has a corresponding Python library). Although it is possible to find a library for almost each data, it is not the only way to process data. An alternative way would have been to write manually every element of a specific format (e.g. for a XML file, with a library, it automatically writes the node '<element>...</element>' by simply calling a method. The alternative was to work only with the strings and write "manually" the XML file).

The visualisation part aims to draw graph on hotspot 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, a python library named tkinter was used which has provided tools to create a graphical user interface. As seen in the validation section, the result (Figure 17) has already many options to customize the graph but it is opened to add more according to the wishes of the user.

The future improvements concerning the data processing part are basically more conversions. In particular, making the conversion in both ways is a feature that may be implemented. For the visualisation part, as already mentioned in section 4.2, the GUI factory can be extended by adding other GUI that can be more efficient and nicer than what tkinter library offers. Another functionality that could have been added is to be able to display more than one graph at a time. Because of the graphical user interfaces strongly depend on the requirements of the user, the number of way to improve the application is extensive.

References

[1] S. cancer screening. https://www.swisscancerscreening.ch/fr/depistage-du-cancer/colon/chiffres-et-faits. Accessed: 2022-11-29.

[2] V. H. Koelzer, I. Zlobec, and A. Lugli, "Tumor budding in colorectal cancer—ready for diagnostic practice?," *Human pathology*, vol. 47, no. 1, pp. 4–19, 2016.

[3] I. Zlobec and A. Lugli, "Tumour budding in colorectal cancer: molecular rationale for clinical translation," *Nature Reviews Cancer*, vol. 18, no. 4, pp. 203–204, 2018.

[4] N. C. Institute, "Stroma definition." https://www.cancer.gov/publications/dictionaries/cancer-terms/def/stroma. Accessed: 2022-12-01.

[5] L. Hogan, "What to Know About Lymph Node Metastasis." https://www.webmd.com/cancer/what-to-know-lymph-nodes. Accessed: 2022-12-02.

[6] N. C. Institute, "T cell definition." https://www.cancer.gov/publications/dictionaries/cancer-terms/def/t-cell. Accessed: 2022-12-01.

[7] 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.

[8] N. C. Institute, "Pathology Reports." https://www.cancer.gov/about-cancer/diagnosis-staging/diagnosis/pathology-reports-fact-sheet. Accessed: 2022-12-03.

[9] 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.

[10] L. Studer, "BT-graph-creation." https://github.com/LindaSt/BT-graph-creation.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] J. Clark, S. DeRose, *et al.*, "Xml path language (xpath)," 1999.

[17] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu, "Xquery 1.0: An xml query language," 2002.

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

[19] "Comma-separated values." `https://en.wikipedia.org/wiki/Comma-separated_values`. Accessed: 2022-11-27.

[20] 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.

[21] "Conda environments." `https://docs.conda.io/projects/conda/en/latest/user-guide/concepts/environments.html`. Accessed: 2022-12-05.

[22] "json — JSON encoder and decoder." `https://docs.python.org/3.9/library/json.html`. Accessed: 2022-12-07.

[23] "lxml.etree." `https://lxml.de/tutorial.html`. Accessed: 2022-12-07.

[24] "xml.dom.minidom — Minimal DOM implementation." `https://docs.python.org/3.9/library/xml.dom.minidom.html`. Accessed: 2022-12-07.

[25] "csv — CSV File Reading and Writing." `https://docs.python.org/3.9/library/csv.html?highlight=csv`. Accessed: 2022-12-07.

[26] "argparse — Parser for command-line options, arguments and sub-commands." `https://docs.python.org/3.9/library/argparse.html`. Accessed: 2022-12-07.

[27] "os — Miscellaneous operating system interfaces." `https://docs.python.org/3.9/library/os.html`. Accessed: 2022-12-07.

[28] "re — Regular expression operations." `https://docs.python.org/3.9/library/re.html`. Accessed: 2022-12-07.

[29] "OpenCV - Open Source Computer Vision." `https://docs.opencv.org/4.5.4/index.html`. Accessed: 2022-12-07.

[30] "Matplotlib: Visualization with Python." https://matplotlib.org/. Accessed: 2022-12-07.

[31] "Numpy - The fundamental package for scientific computing with Python." https://numpy.org/. Accessed: 2022-12-07.

[32] "Graphical User Interfaces with Tk." https://docs.python.org/3.9/library/tk.html. Accessed: 2022-12-07.

[33] 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.

[34] 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.

Appendix

Appendix goes here.