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 purpose, especially the detection of tumour and t-cell in order to identify a potential colon cancer. There are two types

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 is 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 it is scanned.

2.1.1 Whole slide

The whole slide image is the main object, see Figure 1. This is the largest type of data (the size of it is between 90'000 and 200'000 pixels) and it is saved in the format *mrxs*.

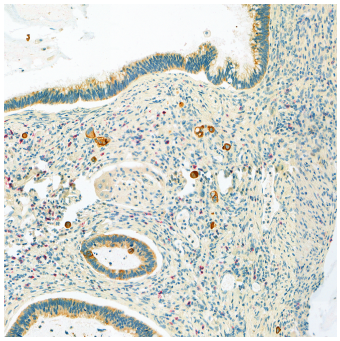
Figure 1: Whole slide image

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* containing annotations and directly draw it on the picture.

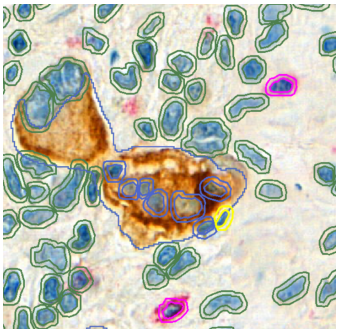
[1] website: `https://computationalpathologygroup.github.io/ASAP/`

2.1.2 Hotspot

Hotspot images are specific area in the whole slide image, hence the size 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]. With this program 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 onto QuPath

Figure 2: Hotspot images

2.1.3 Whole and hotspots combined

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

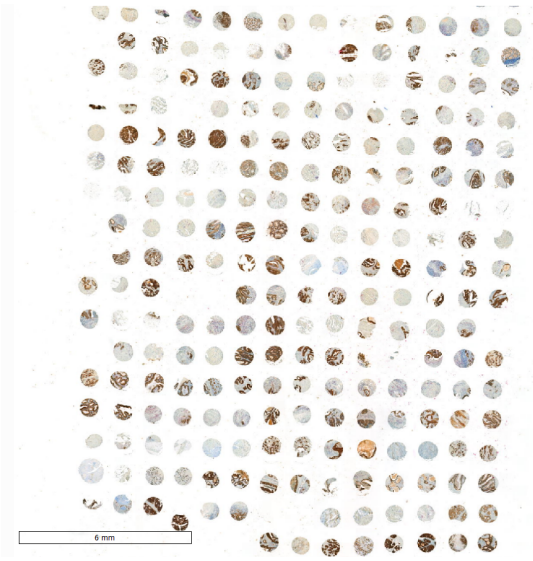


Figure 3: Whole slide and hotpsot combined

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

2.2 Text

Available data are not only pictures but also extra information in textual form. For each hotspot images, there are a *JSON* file associated that contain 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 positions of 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**, which draw the annotations of the hotspot (Figure 2) on the whole slide image (Figure 1). For the coordinates of the annotations to be in the right place, it also need 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 use the ROI (region of interest) points of the *JSON* file. This second conversion is to use 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* 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 the each annotations on the correct ”circle”.

3.2 Resolution

3.2.1 Libraries

As mentioned before in the chapter 1, all programs are implemented in python. First of all, for the three conversion it needs to read *JSON* files, so we must have tools to work with it. Python has a standard library called simply **json**, it allows to open, read and write *JSON* file. It is also necessary to be able to process *CSV* and *XML* files. For *CSV* file, there is also 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** is used. This one allow the user to pass the directory where the files are (or directly file path, it depends how it is implemented or what we need to do) in a command line. The library **os** is also necesseray 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 this 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`

3.2.2 Implementation

First thing to implement is the argument parser that allow use to pass the files in a single command line. With the library **argparse** we can add arguments that force the user to fill them otherwise the program won't compile. 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 that the user write (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, so we need a regular expression that check in the folder which one are of type json and which one are not. Additionally it shouldn't take into account the json files that have a name that is quite different from the other good files (e.g. a good file containing data about hotspot can have the name *Masks_00.2205_1D_AE1_AE3_CD8-level0-hotspot.json*, so if there is another file in the folder having the name *birds.json*, it shouldn't be taken into account because the name is strictly different). With **re** [5] we can create a regular expression that accept the 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*. By using the method search or match. For example, with search method we pass 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 case it will return true as the test string is validate by the regex (note that: **.*** means anything).

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

For the conversion conversion 1 and 3, the output is an xml file that can be read in **ASAP**. So for the both conversions the goal is to construct an **xml tree**. In order to make the xml tree readable by **ASAP**, it must have the following shape:

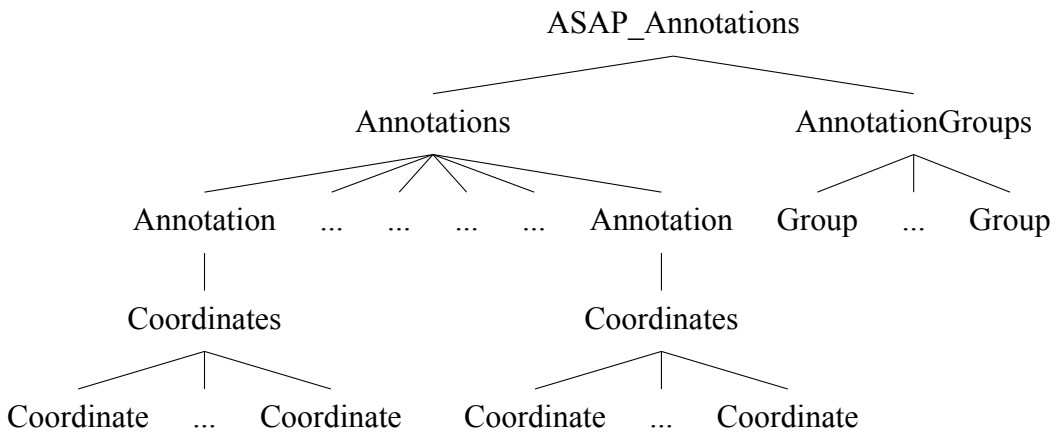


Figure 4: xml tree readable by ASAP

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

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

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

Then for the creation of all the branches, we use the SubElement method where it takes the parent and the name of the new node as parameter. So for the branches **Annotations** and **AnnotationGroups**, it's look like this:

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

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

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

The key *Order* means the number of the current point, i.e. **ASAP** draw the polygon using the classic order, drawing line 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 2 it also need to place the coordinate at the right place on the whole slide image (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 there is an xml file contatining the coordinates of the hotspot. The structure is the following:

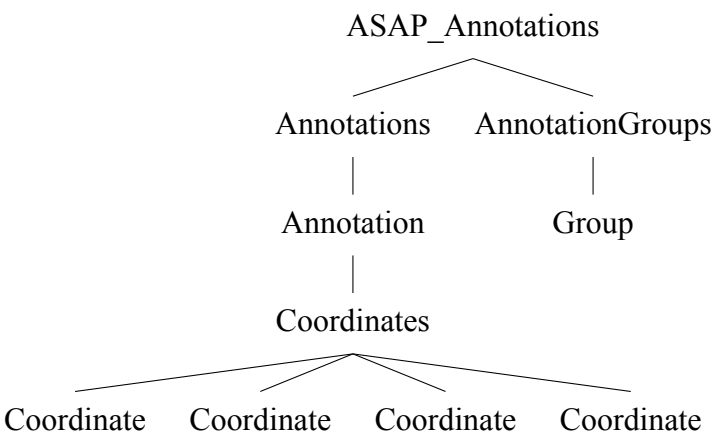


Figure 5: xml tree for the hotspot coordinates

As you may 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 (hotpsot group).

Using **minidom** we can get the values of the elements **Coordinate** with the method *getElementsByTagName* and then we can get easily the attributes 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]
```

Since we get the coordinates of the hotspot we can draw it and move the ROI points inside of it.

For the second conversion, this time there is a csv file containing coordinates of the hotspot. The file 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 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 good coordinates we only have to add the center coordinates to the ROI points and subtract the radius.

For the last conversion (2) the strategy for saving features on a csv file is to create a dictionary for each row and write it.

```
    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 file we extract the interested features and save them on a dictionary that is used to write a row in the csv file.

3.2.3 Validation

For the first conversion (1), as explained before, the xml file generated by the script can be loaded in **ASAP**. So we can easily check the validity of the application, by loading the corresponding file to the right 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 on 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 output xml file it draws the annotation on the specific hotspot.

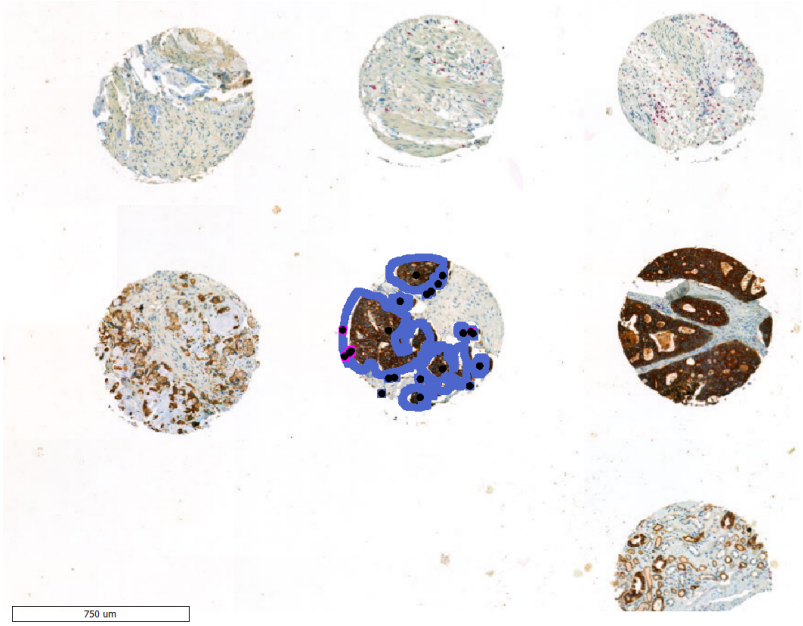


Figure 9: Drawing of the circular hotspot

4 Visualisation

4.1 Problem statement

The visualisation part means 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 coordinates. This other file is of type *gxl*, which is a format similar 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.

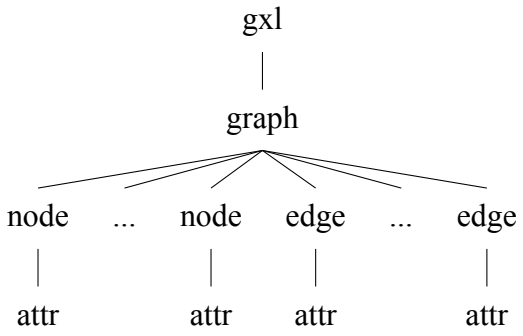


Figure 10: gxl tree

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

The application should have 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, it directly update the graph.

4.2 Resolution

4.2.1 Libraries

Similar to the strategy of the data processing part (section 3.2) visualisation also need libraries as **os** (for the processing of every file/folder path for input/output), **re** (find the proper gxl file corresponding to the image given) 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 main library is **tkinter** [7]. With this tool, it is possible to create button, scrollbar, display picture in a box, ... 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.

4.2.2 Implementation

The first part of the implementation is to allow the user to be able to pass an image folder and a folder of gxl files. 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** it is possible to create 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 press 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 path given. You can also notice that the position of the button on the grid that is (0,0) with a columnspan of 2. It means that the position is top left and takes two columns of space (same as *button1* on Figure 11) .

For the matching of gxl file and image, again it needs 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 it will draw the corresponding image on the picture box. In order to find the image corresponding to a gxl file, firstly we extract the name of the gxl file, then we search for the image name that contains it. So the regular expression would be construct 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`

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. So we need to create interface for this purpose. For the choice of a color, **tkinter** provide a very nice tool that is *colorchooser*. It can ask the user for a color and display this interface:

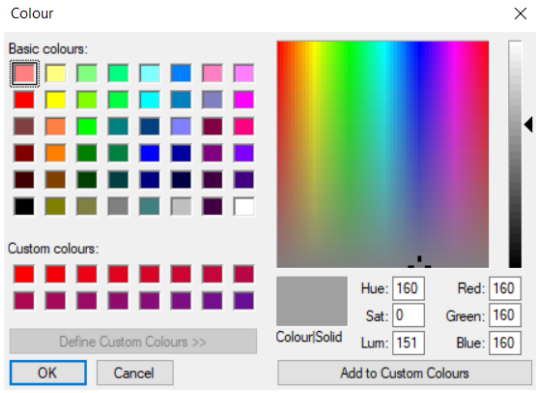


Figure 12: colorchooser

When the user select a color, it returns it on the format RGB and also in HEX code.

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

Now that the basic configuration is in place, it is possible to draw the graph on the image. As said in section 4.2.1, 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 *img* 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).

For the drawing of the edges, it is quiet similar to the nodes. This time it needs 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'],
2                           lineType=self.edge_style['lineType'])
```

Assume that we draw all the nodes first, so 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.

4.2.3 Validation

TODO

5 Conclusion

TODO

References