TP : path finding from a plan

In this project, we will analyze a plan in order to extract a navigation map, then we will implement and compare several ways to find the shortest path between two points in the plan. Finally, we will use Pattern Matching to find way points defining a tour.

I Image Binarization

1) Get the image file at this address: https://github.com/gaysimon/TP_pathfinding. You will first write a program to display this image on screen.

Tip: *- load an image and convert to matrix in Java*

```
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
//////////////////////
BufferedImage image;
try {
       image = ImageIO.read(new File("/path/to/the/image.png"));
} catch (IOException e) {System.out.print(e);}
//////////////////////
map=new int[image.getWidth()][image.getHeight()][3];
for (int i=0;i<image.getWidth();i++){
       for (int j=0;j<image.getHeight();j++){
              map[i][j][0]=(image.getRGB(i, j)>> 16) & 0x000000FF;
              map[i][j][1]=(image.getRGB(i, j)>>  8) & 0x000000FF;
              map[i][j][2]=(image.getRGB(i, j)     ) & 0x000000FF;
       }
}
```

- Load, read and write an image in Python (you can also use cv2 *if* openCV *is installed)*

```
import matplotlib.image as mpimg
######################
img = mpimg.imread("my/image.png")
if img.dtype == np.float32: # if you want to convert values to [0;255] integer
    img = (img * 255).astype(np.uint8)
######################
r=img[50,50,0]
g=img[50,50,1]
b=img[50,50,2]
######################
img[50,50,0]=r
img[50,50,1]=g
img[50,50,2]=b
```

2) The image has some problems that make it unsuitable for automatic processing: gray pixels due to image compression, color icons and texts that must be removed. Write a program to *binarize* the image and obtain a black or white image showing walls

Reminder: the binarization operator consists of applying thresholds (or any boolean test) on each pixel of the image. As we use a RGB color image, you will have to threshold the three channels (red, green and blue). Here, the walls have a color of (217, 208, 201) in RGB space (the color may however be slightly altered by image compression, so use margins in your threshold values).

3) We want to add a security distance of 5 pixels around the walls. Create a second binary image by using a *dilation* operator on the first binary image, with a square kernel of 11x11 pixels. Then display this security distance with a different color than walls.

Reminder: the dilation is a convolution with a mask on a binary image. When a pixel of the input image is 1, the mask is added to the output image. The inverse operator of dilation is the *erosion.*

Figure 1: left: initial image (1), middle: binary image (2) displaying walls,
right: image displaying security distances in green.

II Using a path finding algorithm on a map

Path finding consists in defining the shortest possible path between two points on a map with obstacles. This principle is often used in robotics, but also in videogames to move NPCs in the environment. We will implements and compare three algorithms: graph traversal, A* and WA* (Weighted A*).

The graph traversal algorithm consists in defining the minimum distance from the initial point at each accessible point of the environment. Then, it is possible to define the shortest path to any point of the environment.

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Figure 2 : execution of a path finding algorithm in a discrete environment.
Each cell contain the minimum distance from start point (Manhattan distance).

1) Implement the graph traversal described below :

Initialization :
define *point_start* // *select two points of your choice*
define *point_end* // *in empty areas of the image*

```
create an empty list list
create a matrix M with the same dimensions than the image
initialize M with value INFINITY                          // you can use 1,000,000 as INF

set M[point_start] to 0
add point_start to list

main loop
while list is not empty do
    p0 = list[0]
    for each of the 4 accessible neighbors pi of p0 do    // i.e. in image AND not a wall
            if M[p0] + 1 < M[pi] then
                    M[pi]=M[p0]+1
                    add pi to list
            end if
    end for
end while
```

At the end of this program, you will obtain a matrix M giving the minimal distance of each point of the image (from the start point).

2) Modify your program to display the distance map, with the color of the pixel giving its distance from the start point. Example : color=(1-(*a**dist(p) , 0 , *a**dist(p)) (color change from red to blue when distance increase, a is a coefficient)

3) Define the shortest path between your two points, by completing your program with the following algorithm:

```
create an empty list path
if M[point_end]==INFINITY then write “the destination cannot be reached”
else
    p0=point_end
    add p0 to path
    while p0!=point_end do
            select pi_min among the 8 neighbors pi of p0 where M[pi] is minimal
            p0=pi_min
            add p0 to path
    end while
end if
```

After the execution of your program, the *path* list contains the sequence of points forming the shortest path between point_start and *point_end*.

4) Display the path on your map with the color of your choice

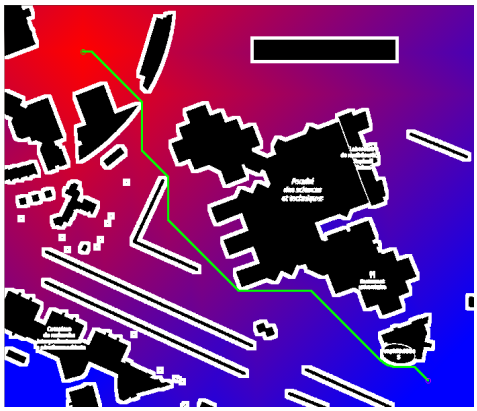

Figure 3: map of distances and path, from start point (top left) to end point (bottom right).

III The A* algorithm

The graph traversal algorithm has a serious disadvantage: it is long to execute, especially with grids of large size. It is thus not very appropriate in applications that require a strong responsiveness. It is however possible to optimize this algorithm by adding a *Heuristic* (simplification hypothesis). In this case, we will define a priority to points according to their distance from start and ending points. The hypothesis is that the shortest path is the straight line. We thus define a *value of heuristic* of a point p as :

$$h_p = \text{dist(p, point\_start)} + \text{dist(p, point\_end)}$$

which correspond by the minimal possible length of a path passing through this point. The list of points *list* is ordered by increasing heuristic values, such as the points that may be part of the closest path are tested first. The algorithm can stop when the destination *point_end* is reached.

1) create a new function that implements the A* algorithm described below :

```
Initialization :
define point_start                                  // select two points of your choice
define point_end                                    // in empty areas of the image

create an empty list list
create a matrix M with the same dimensions than the image
initialize M with high value INFINITY                       // you can use 1 000 000 as INF

set M[point_start] to 0
add point_start to list
p0 = list[0]

main loop
while list is not empty AND list[0] != point_end do
    p0 = list[0]
    for each of the 4 accessible neighbors pi of p0 do      // i.e. in image AND not a wall
            if M[p0] + 1 < M[pi] then
                    M[pi]=M[p0]+1
                    pi.h = M[pi] + dist(pi, point_end)
                    insert pi in ordered list
            end if
    end for
end while

path finding
if M[point_end]==INFINITY then write “the destination cannot be reached”
else
    create an empty list path
    p0=point_end
    add p0 to path
    while p0!=point_end do
            select pi_min among the 8 neighbors pi of p0 where M[pi] is minimal
            p0=pi_min
            add p0 to path
    end while
end if
```

2) Display the distance map and the path on the image. What do you observe when compare with the previous method?

3) Answer in your report: what are the pros and cons of each method?

IV The Weighted A* algorithm

The A* algorithm has the advantage of being both fast and reliable. But some variations tries to speed up the execution, by modifying the heuristic value equation. One of the most famous is the Weighted A*. This algorithm modify the heuristic value by giving more importance to the distance between a point and the destination point. The heuristic is then defined as :

$$h_p = \text{dist}(p, \text{point\_start}) + W \times \text{dist}(p, \text{point\_end})$$

Were W in [1; +inf[is the weight. Note that if W=1, the algorithm is the basic A*. This simple modification can significantly speed up the execution of the algorithm, but has some disadvantages.

1) Write a new function, based on the previous one, to implement the WA*. You will test it with several values of W (e.g. 1.5, 2, 5, 10) and with several points.

2) Execute the three algorithms on the same pair of point and display the three paths. Try with several points. What do you observe?

V Pattern Matching

At this address: https://github.com/gaysimon/TP_pathfinding, you can download a set of four small images that are part of the map. These images indicate four way points that you must visit to complete a tour. You first have to detect these points in the map.

When the patterns are very close to the original image (no rotation, no scaling), a pattern matching can be applied to find the position of this pattern in the image. Note that otherwise, feature matching algorithms (using descriptors methods like SURF, SHIFT or ORB) are more suitable.

Reminder: pattern matching consists of comparing each pixel of the pattern with a part of the image around the currently tested position. This algorithm can thus be rather long with large patterns.

1) Load the four patterns, then implement the following pattern matching algorithm to get the position of the four way points. Note that patterns have a size of 40 x 40 pixels. You will display the four way points on the image with circles.

```
get image img
get pattern pattern
create a matrix M with the same dimensions than the image              // matching matrix

Compute matching
for x in [ 0; size_X – 40 [ do                                  // for each position (x,y) of img
        for y in [ 0; size_Y – 40 [ do
                sum = 0
                for i in [ 0; 40 [ do                           // for each pixel (I,j) of pattern
                        for j in [ 0; 40 [ do
                                sum += | pattern[i][j] – img[x+i][y+j] |        // absolute value
                        end for
                end for
                M[x+20][y+20] = sum / 1600                      // 40 x 40 = 1600 pixels
        end for                                                 // don't forget that pattern center is at (20;20)
end for

Get the minimum difference
set min to INFINITY
set x_min to -1
set y_min to -1
for x in [20; size_X - 20[ do                                   // margin of 20 px around image
        for y in [20; size_Y - 20] do                           // due to pattern size
                if ( M[x][y] < min do
                        min=M[x][y]
                        x_min=x
                        y_min=y
                end if
        end for
end for
store way point ( x_min; y_min )
```

Tip: *- you can speed up the execution by testing only one pattern pixel on two for line and for column, reducing by four the number of pixel (you will then divide sum by 400)*

2) Using your A* function, compute the complete route visiting the four way points and display it on the map.

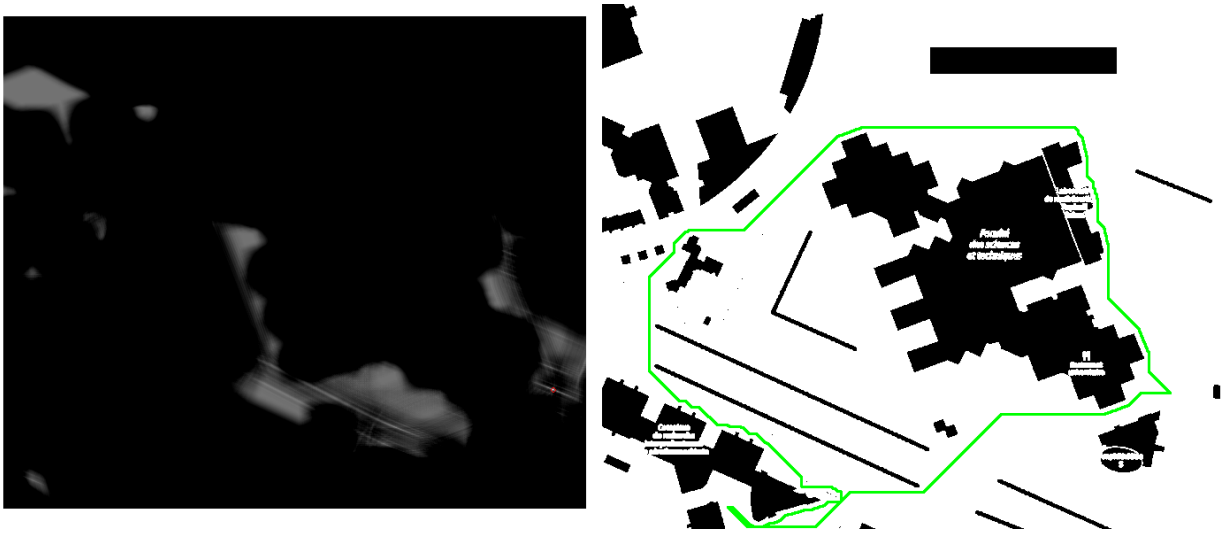

Figure 4: left: matching map for the first pattern. Bright indicate low difference. The red circle indicates the best matching. Right: the full route, passing through way points