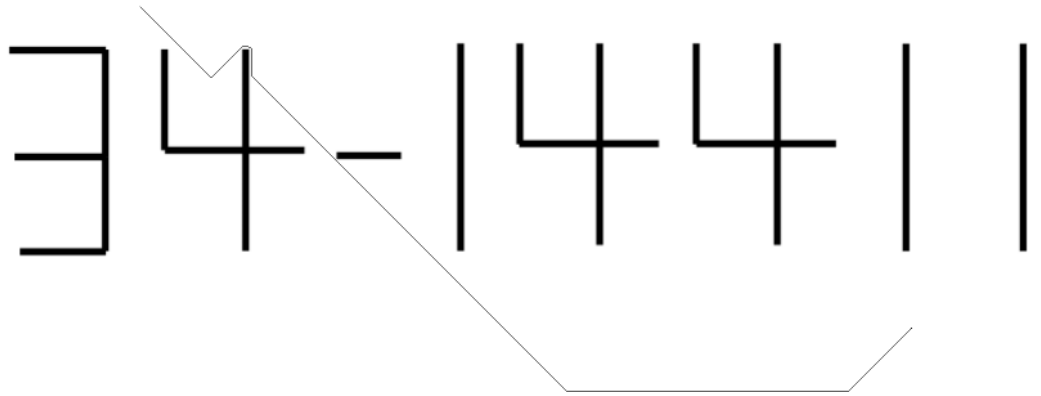
Report for Breadth-First and Depth-First Searching Algorithms Implementation Using Python Functions

Breadth First Algorithm:

Two Breadth-first functions have been implemented. The **first** function is ***"breadth_first (map, row_init, col_init, row_goal, col_goal)"*** . The function takes 5 variables, 1st is map/2D array which is converted from an image, the array contains values ranging from '0' which is black to '255' which is white. The 2nd and 3rd variables are to define x and y coordinated in the array for the initial point/ current point. Finally, the 4th and 5th variables are for the coordinates for the goal point. The functions outputs the path in the terminal when run and saves an image "output.png" showing the initial and final points as light gray and black dots respectively and also shows the path taken in gray dots. The terminal outputs the path taken to reach goal in the form of points "[(x,y),(…),...]"
starting from initial point to goal point.

Examples:

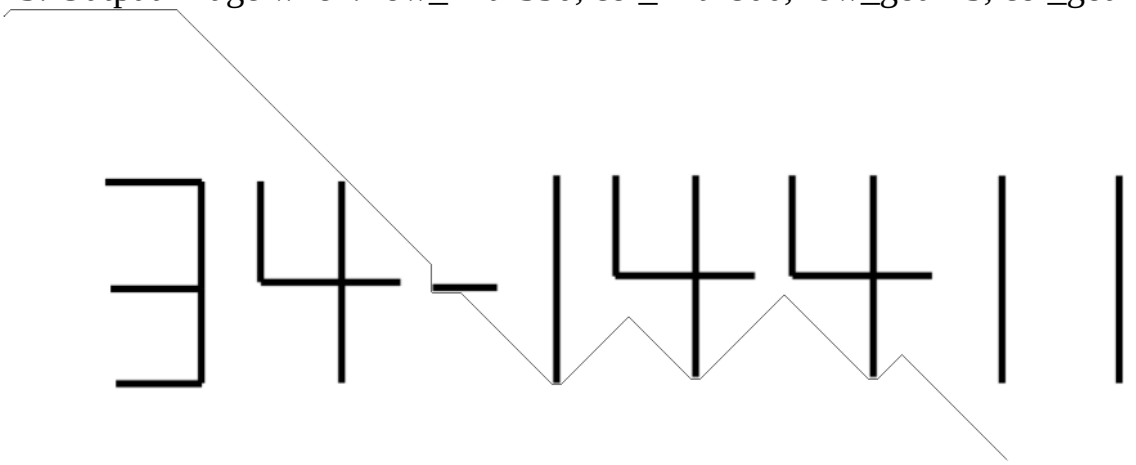
Ex1.Output image when: row_init=100, col_init=200, row_goal=350, col_goal=800



As you can see above the path is drawn on free space from start point in top left to goal point in bottom right avoiding obstacles.

Ex2.Output image when: row_init=100, col_init=200, row_goal=250, col_goal=250
This is similar to previous example except goal is closer to initial point.

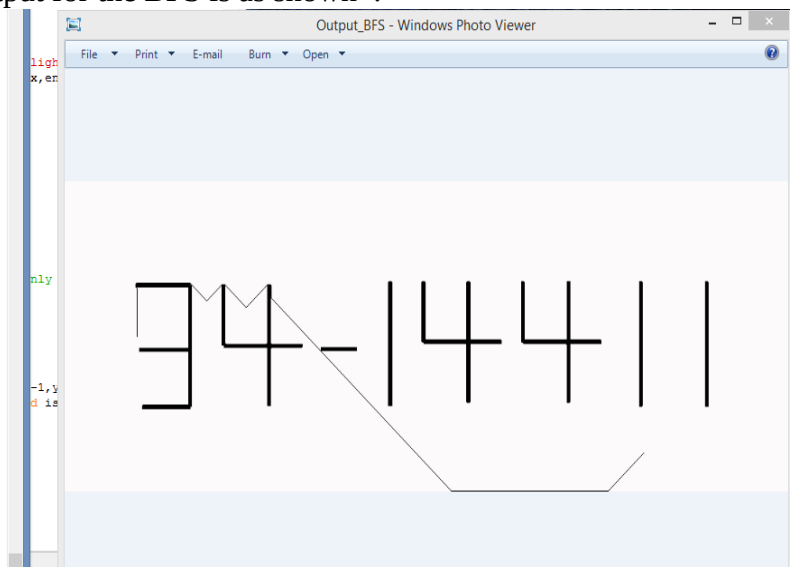
Ex3. Output image when: row_init=350, col_init=800, row_goal=5, col_goal=20



Here the start is at the bottom right and the finish is at the top left

The **second** breadth-first function is also by loop on the queue , but different , by adding external methods , to check if the observed cell wall cell , or obstacle cell ,etc. , and the code can be used as the following , it is developed to work and outputs an image called "Output_BFS.png" , but differs it uses an ordered exploration direction starting from the north direction of the robot , then in direction of anti-clock wise , for example : [North , North West , West , South West, South , South East , East , North East] , and then put into the queue then by restarting the loop (iterating) , the queue outs according to (First in , First Out) [FIFO] algorithm , and by applying it , it first gives the goal another value in the array , which differs from 0 and 1 , that is only to highlight the goal for the robot , (for us as used in tracing) , and the output for the BFS is as shown :

```
move in j : 382 then move in i : 768
move in j : 381 then move in i : 769
move in j : 380 then move in i : 770
move in j : 379 then move in i : 771
move in j : 378 then move in i : 772
move in j : 377 then move in i : 773
move in j : 376 then move in i : 774
move in j : 375 then move in i : 775
move in j : 374 then move in i : 776
move in j : 373 then move in i : 777
move in j : 372 then move in i : 778
move in j : 371 then move in i : 779
move in j : 370 then move in i : 780
move in j : 369 then move in i : 781
move in j : 368 then move in i : 782
move in j : 367 then move in i : 783
move in j : 366 then move in i : 784
move in j : 365 then move in i : 785
move in j : 364 then move in i : 786
move in j : 363 then move in i : 787
move in j : 362 then move in i : 788
move in j : 361 then move in i : 789
move in j : 360 then move in i : 790
move in j : 359 then move in i : 791
move in j : 358 then move in i : 792
move in j : 357 then move in i : 793
move in j : 356 then move in i : 794
move in j : 355 then move in i : 795
move in j : 354 then move in i : 796
move in j : 353 then move in i : 797
move in j : 352 then move in i : 798
move in j : 351 then move in i : 799
move in j : 350 then move in i : 800
Goal Reached ! , Time taken to find goal : 5.3613505363464355
Path Drawn from start to goal which cell value is '8'
Path saved to 'Output_BFS.png' in same Dir of .py file If On Windows
```



the code sequence the goal is set , and start point then function called "implement_BFS(my_start,my_goal)" and this method uses a helper method that returns list of paths taken to reach goal.

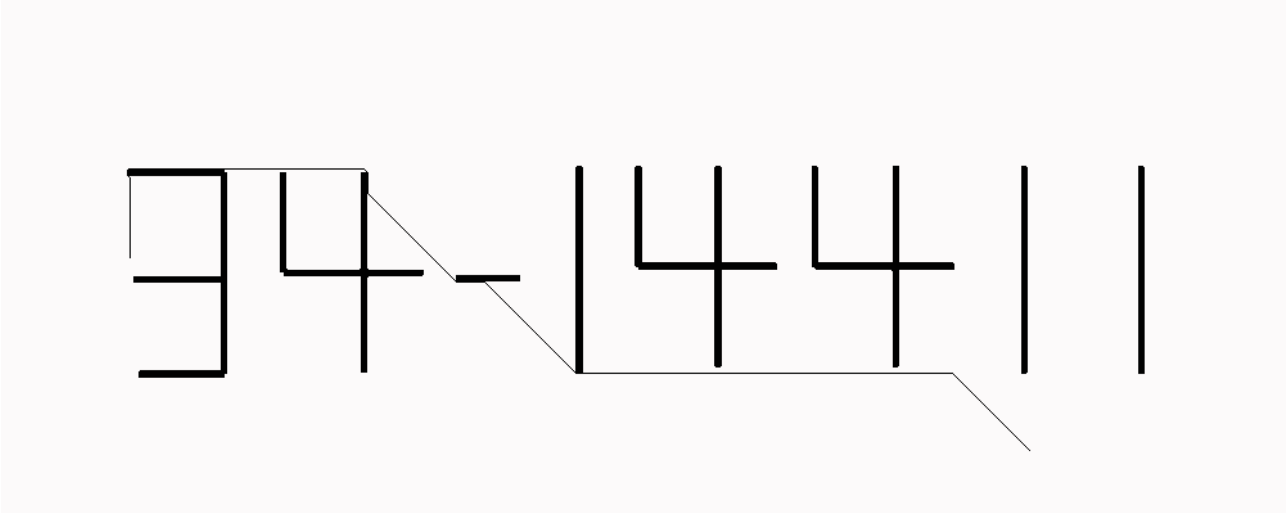
```
BFSstart = time.time() #  <========= capture time before execution

my_start = [350, 100] # giving start position j,i    <=======================================    start point
my_goal = [30, 800] # giving end position    <=======================================    goal point

path_arry_list = implement_BFS(my_start,my_goal) #<================================== needed BFS method takes start and goal points only

print(path_arry_list)
length = len(path_arry_list)
print('============== Given Path ==============')
for ipj in range(length):
    x_path = path_arry_list[ipj][0]
    y_path = path_arry_list[ipj][1]
    matrix_outx[y_path,x_path] = 0
    print(" move in j : " +str(y_path) + " then " + "move in i : " +str(x_path) )

BFSend = time.time()#  <========= capture time after execution
```

Ln: 136 Col

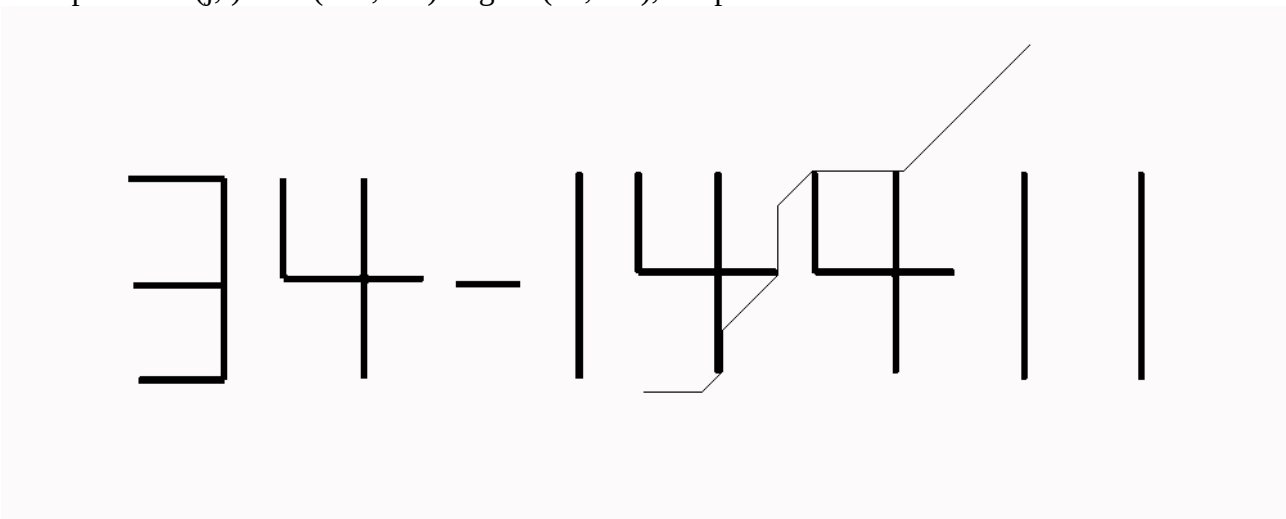
```
def implement_BFS(start_point=[],goal_point=[]):
    global matrix
    var_queue = queue.Queue()
    start_x = start_point[0]
    start_y = start_point[1]
    end_x = goal_point[0]
    end_y = goal_point[1]
    matrix[end_x,end_y] = 8 # <===== give a flag / Higlight the goal point (enha teba 8 fe west el 0s we el 1s)
    return bfs_helper(matrix, (start_x, start_y),(end_x,end_y))


for iter in range(1):
    start_x,start_y = 0,0 #
def bfs_helper(grid, start,goalp):
    global matrix
    queue = collections.deque([[start]])
    goal = matrix[goalp[0],goalp[1]]
    height = len(matrix)
    width = len(matrix[0])
    print("Usinf BFS : " + str(width))
    print("Goal cell changed from 1 to value '8' for only Highlighting it => " + str(goal))
    seen = set([start])
    while queue:
        path = queue.popleft()
        x, y = path[-1]
        if grid[y][x] == goal:
            return path
        for x2, y2 in ((x,y-1) , (x-1,y-1), (x-1,y),(x-1,y+1),(x,y+1),(x+1,y+1),(x+1,y),(x+1,y-1)):
            if 0 <= x2 < width and 0 <= y2 < height and is_wall_cell(x2,y2)==False and is_obstacle(x2,y2)==False and (x2, y2) not in seen:
                queue.append(path + [(x2, y2)])
                seen.add((x2, y2))
```

the main method takes the points (x_start,y_start) (x_desired,y_desired) , and then it prepares the points inform of readable integer pointing the start point and other for goal point , then by setting 5 start points and end point

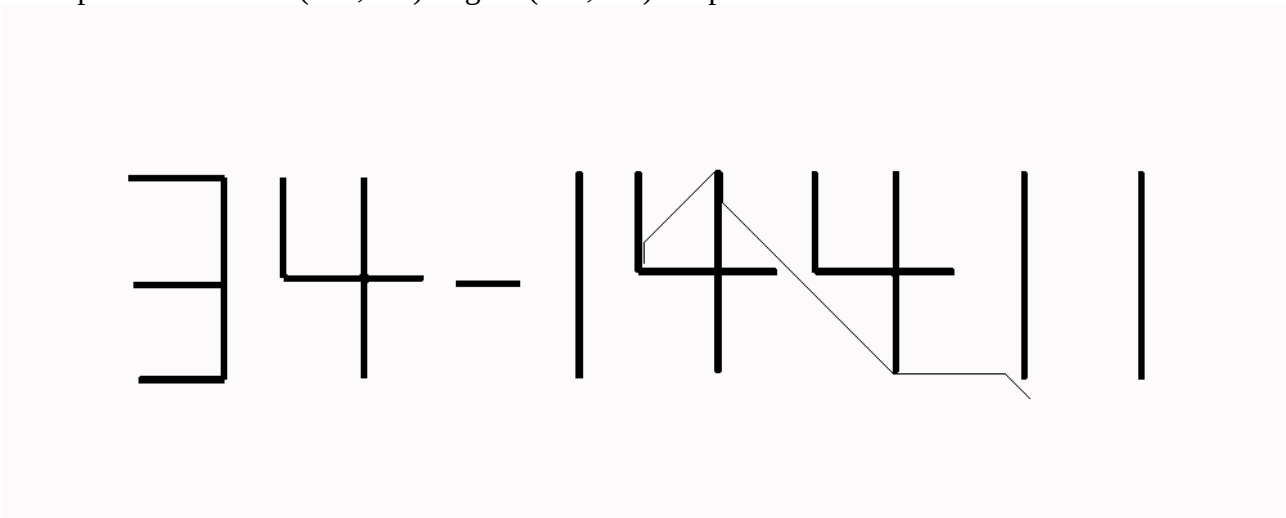
Example 1 : from start (100,200) to goal (350,800) the path outs as shown



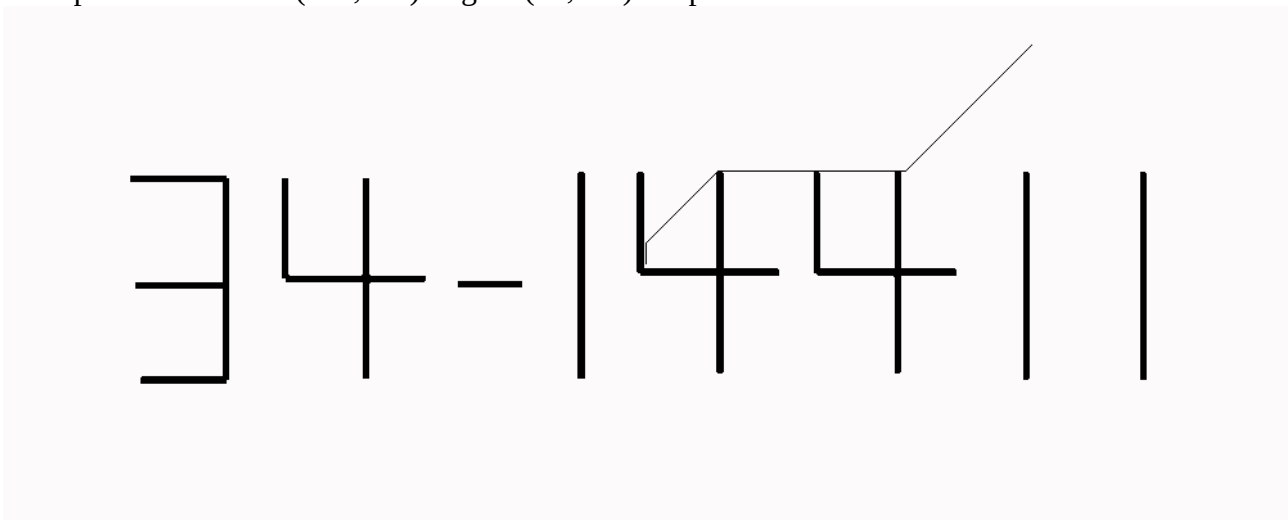
Example 2 : as (j,i) start (500,300) to goal (30,800), the path outs as shown



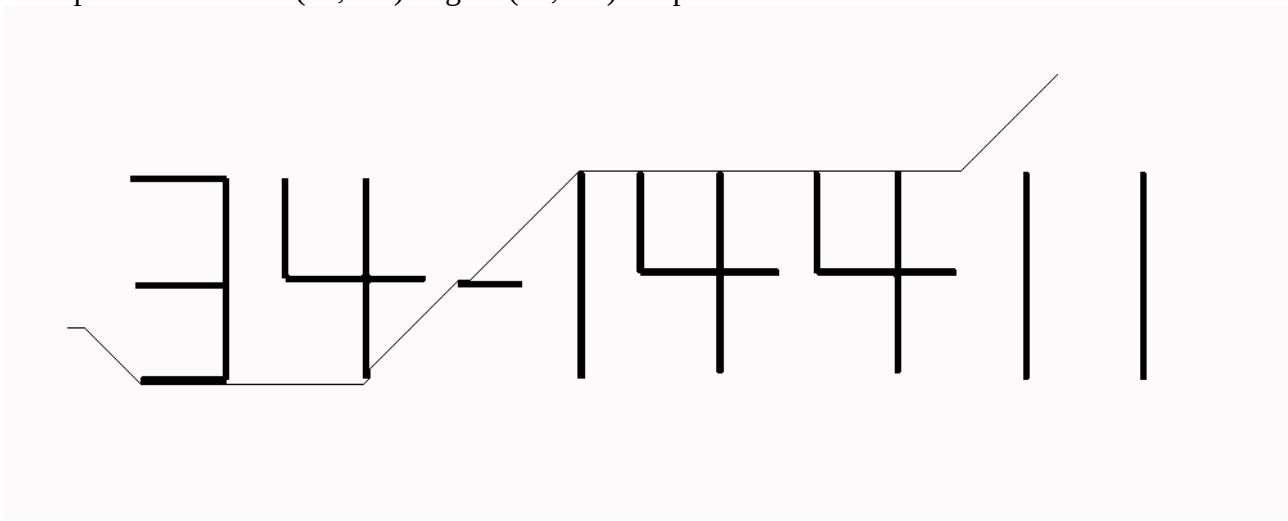
Example 3 : from start (500,200) to goal (305,800) the path outs as shown



Example 4 : from start (500,200) to goal (30,800) the path outs as shown



Example 5 : from start (50,250) to goal (53,820) the path outs as shown



Depth First Algorithm :

(DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

The DFS depends on filling a stack that follows (Last in - First Out) , and also uses a helper method as shown

mycode.py - C:\Users\Admin\Autonomous\Quiz1\mycode.py (3.8.2)

File Edit Format Run Options Window Help

```
def implement_DFS(start_point=[],goal_point=[]):
    global matrix
    start_x = start_point[0]
    start_y = start_point[1]
    end_x = goal_point[0]
    end_y = goal_point[1]
    matrix[end_x,end_y] = 8 # <===== give a flag / Higlight the goal point (enha teba 8 fe west el 0s we el 1s)
    return dfs_helper(matrix, (start_x, start_y),(end_x,end_y))

def dfs_helper(grid, start,goalp):
    global matrix
    stack =  ([[start]])
    goal = matrix[goalp[0],goalp[1]]
    height = len(matrix)
    width = len(matrix[0])
    print("Using DFS : " + str(width))
    print("Goal cell changed from 1 to value '8' for only Highlighting it => " + str(goal))
    visited = set([start])
    path = [start]
    reached = False
    while stack:
        path = stack.pop(0)
        #print("here1:" + str(path))
        x, y = path[-1]
        #stack.append(path + [(x,y)])
        if grid[y][x] == goal or (y==goalp[0] and x ==goalp[1]):
            reached = True
            return path
        for x2, y2 in ((x,y-1) , (x-1,y-1), (x-1,y),(x-1,y+1),(x,y+1),(x+1,y+1),(x+1,y),(x+1,y-1)):
            if 0 <= x2 < width and 0 <= y2 < height and is_wall_cell(x2,y2)==False and is_obstacle(x2,y2)==False and (x2, y2) not in visit
                #print("here2: "+ str(path))
                stack.append(path + [(x2,y2)])
                visited.add((x2, y2))
```

the function can be called "implement_DFS(my_start,my_goal)" as shown below

```
BFSstart = time.time() #  <========= capture time before execution

my_start = [50, 250] # giving start position i,j    <====================================    start point
my_goal = [53, 820] # giving end position    <====================================    goal point

path_arry_list = implement_BFS(my_start,my_goal) #<================================== needed BFS method takes start and goal points only
print(path_arry_list)
length = len(path_arry_list)
print('============== Given Path ==============')
for ipj in range(length):
    x_path = path_arry_list[ipj][0]
    y_path = path_arry_list[ipj][1]
    matrix_outx[y_path,x_path] = 0
    print(" move in j : " +str(y_path) + " then " + "move in i : " +str(x_path) )

BFSend = time.time()#  <========= capture time after execution
print ("Goal Reached ! , Time taken to find goal : " + str(BFSend-BFSstart))
np.set_printoptions(threshold=np.inf, linewidth=np.inf)

im = Image.fromarray(np.uint8(cm.gist_earth(matrix_outx)*254))
im.save("Output_BFS.png")
#=======================================DFS=================================
with open('path_out_BFS.txt', 'w') as f:
    f.write(np.array2string(matrix_outx))
print ("Path Drawn from start to goal which cell value is '8'")
print ("Path saved to 'Output_BFS.png' in same Dir of .py file If On Windows")
matrix_outx = matrix
DFSstart = time.time() #  <========= capture time before execution
path_arry_list = implement_DFS(my_start,my_goal) #<================================== needed DFS method takes start and goal points only

print(path_arry_list)
length = len(path_arry_list)
print('============== Given Path ==============')
for ipj in range(length):
    x_path = path_arry_list[ipj][0]
    y_path = path_arry_list[ipj][1]
    matrix_outx[y_path,x_path] = 0
    print(" move in j : " +str(y_path) + " then " + "move in i : " +str(x_path) )

DFSend = time.time()#  <========= capture time after execution
print ("Goal Reached ! , Time taken to find goal : " + str(DFSend-DFSstart))
np.set_printoptions(threshold=np.inf, linewidth=np.inf)
```

Ln: 156 Col: 40

and both of them are implemented in the same code (mycode.py) as shown then outs , the output png images one for the BFS and other DFS , but the input we converted it after drawing the image , we converted it to

"id_array.txt" into 0 & 1 array and the code first convert it to a valid 2d matrix (2d array of integers) of 0 & 1 , that text file is accompanied by the python file in same directory , and starts reading it line by line then char by char as shown in the image below

mycode.py - C:\Users\Admin\Autonomous\Quiz1\mycode.py (3.8.2)

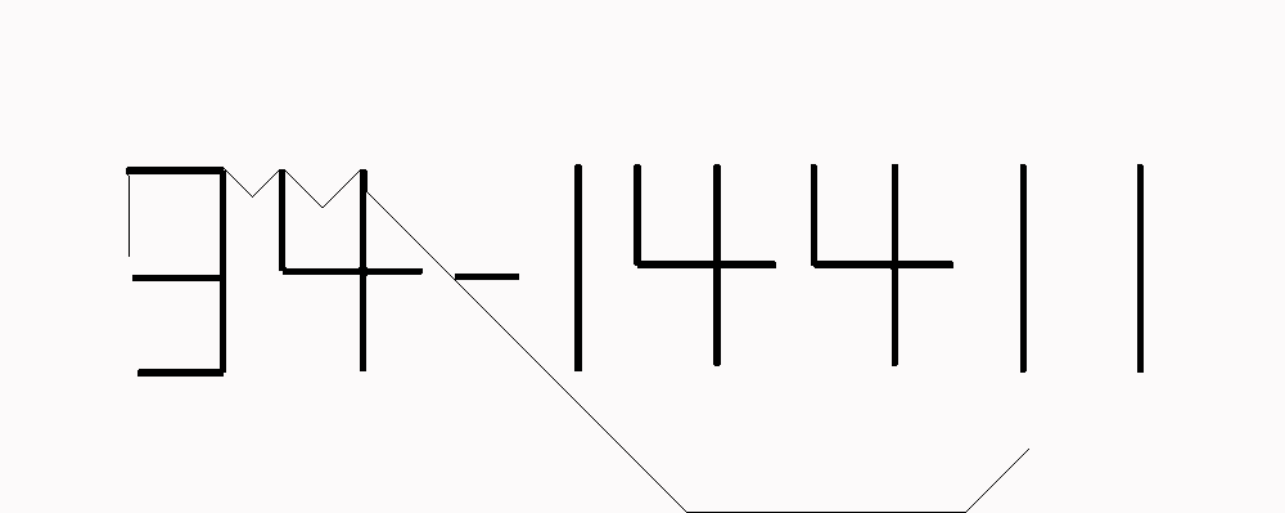
File Edit Format Run Options Window Help

```
import numpy as np
import queue
import time
import sys
import os
import collections
from matplotlib import cm
from PIL import Image

i = -1 # are horizontal
j = 0  # are vertical
value = 0
matrix = np.zeros((400,1000)) # make empty zeros 2D integer Array
matrix_outx = np.zeros((400,1000)) # make empty zeros 2D integer Array for output
with open('id_array.txt', 'r') as f:
    for line in f: #loop for each line
        i = -1
        for c in line: #loop for each char in each line
            if c=='\n': #decrement the j again to avoid size increasing
                j = j + 1
                i = i - 1
            if c=='1': # if the char is 1 edits the value in the empty 2d arrays
                i = i + 1
                matrix[j,i] = 1 # if the char is 1 edits the value in the empty 2d arrays
                matrix_outx[j,i] = 1  # if the char is 1 edits the value in the empty 2d arrays
            if c=='0': # if the char is 0 edits the value in the empty 2d arrays
                i = i + 1
                matrix[j,i] = 0 # if the char is 1 edits the value in the empty 2d arrays
                matrix_outx[j,i] = 0# if the char is 1 edits the value in the empty 2d arrays


np.set_printoptions(threshold=np.inf, linewidth=np.inf)  # turn off summarization, line-wrapping
with open('output2D_Valid_Arry.txt', 'w') as f:
    f.write(np.array2string(matrix))
obstacles = ""
explored = ""
def is_obstacle(Cell_X,Cell_Y):
    if Cell_X <= 999 and Cell_Y <= 399 :
```

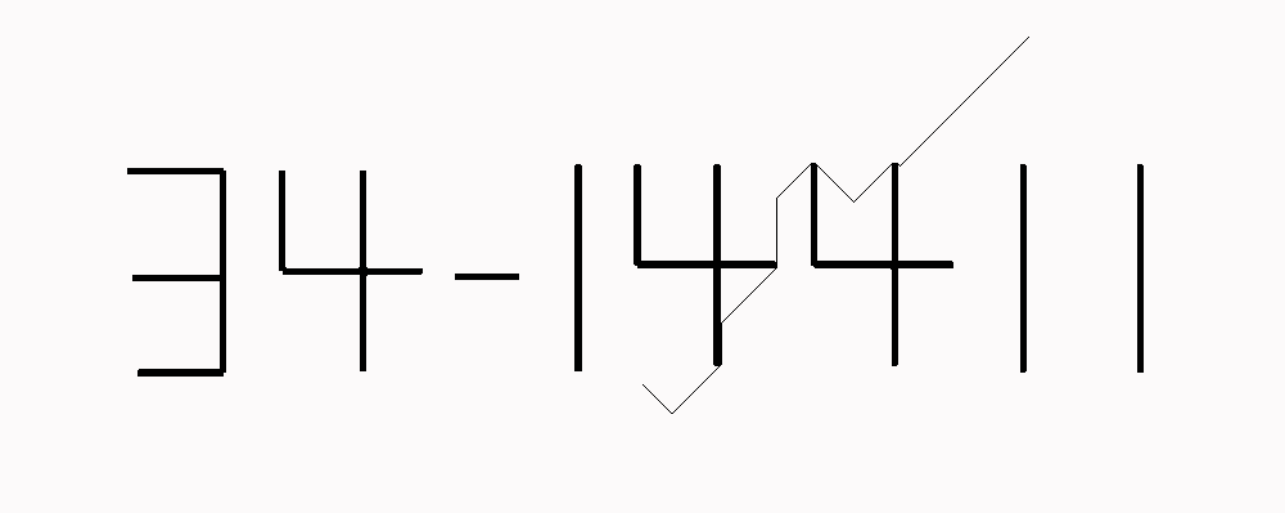
then outs variable called "matrix" which is a valid 2d array to work with , and variable "matrix_outx" is another copy of it , which is used to mark the output path , then write it into png image as shown below

```
im = Image.fromarray(np.uint8(cm.gist_earth(matrix_outx)*254))
im.save("Output_BFS.png")
#=========================================DFS=====================================
with open('path_out_BFS.txt', 'w') as f:
    f.write(np.array2string(matrix_outx))
print ("Path Drawn from start to goal which cell value is '8'")
print ("Path saved to 'Output_BFS.png' in same Dir of .py file If On Windows")
```

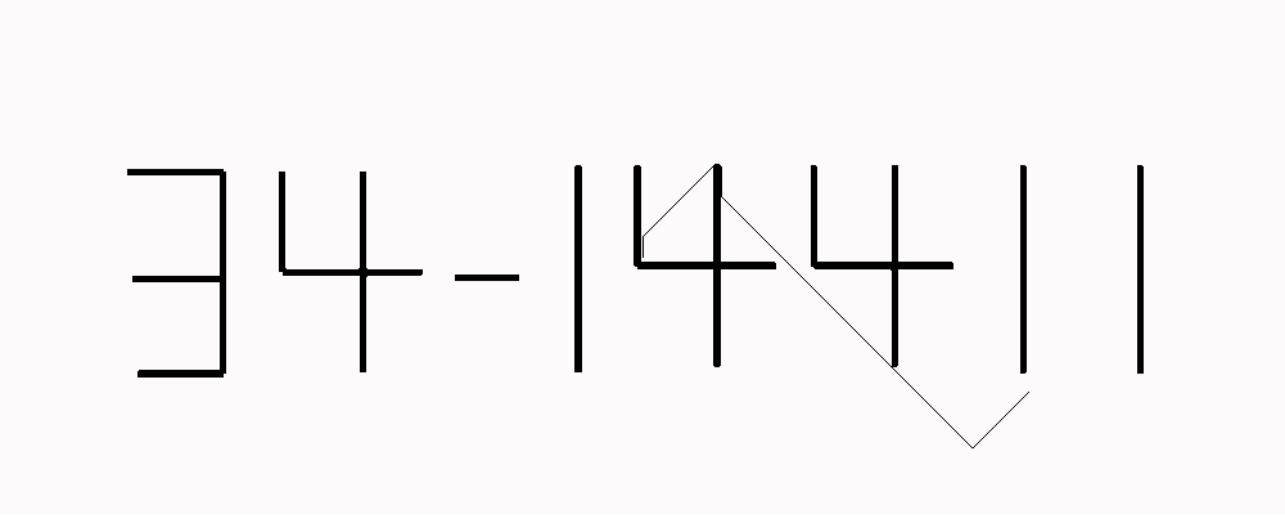
By applying the developed DFS part at the same python file "Output._DFS.png"
same goal and start , but the output line is in a zig-zag form, by applying same examples (same start and goal points)
Example 1 : from start (100,200) to goal (350,800) the path outs as shown



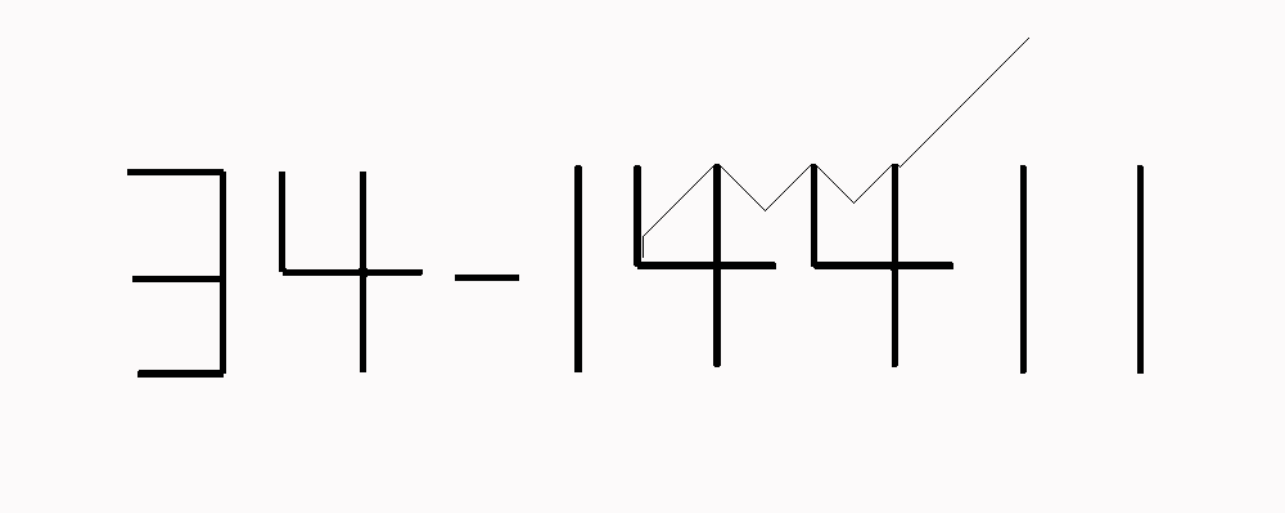
Example 2 : from start (500,300) to goal (30,800) the path outs as shown



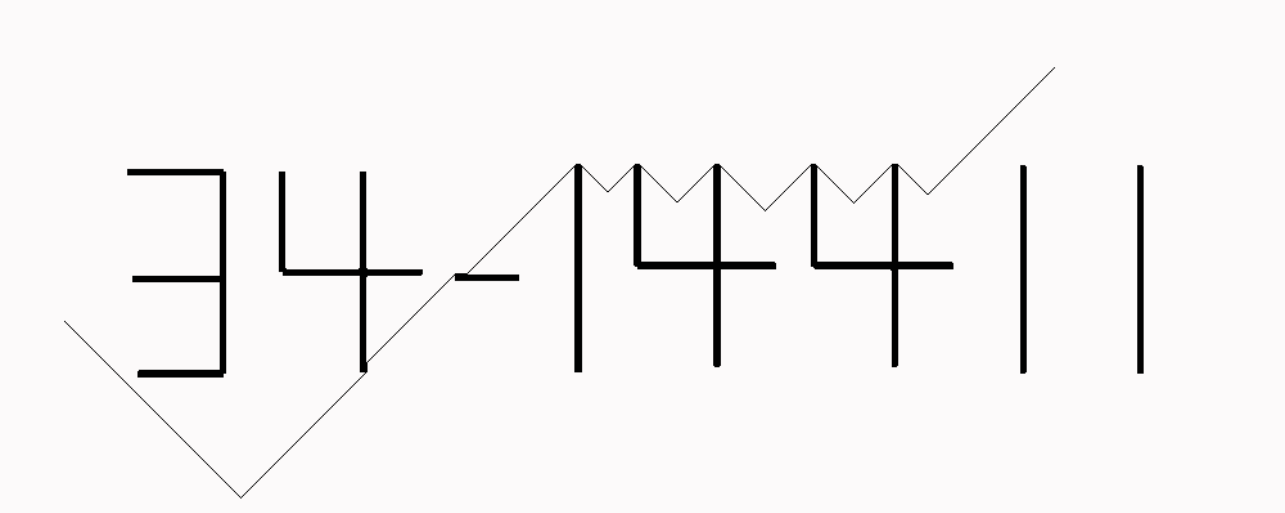
Example 3 : from start (500,200) to goal (305,800) the path outs as shown



Example 4 : from start (500,200) to goal (30,800) the path outs as shown



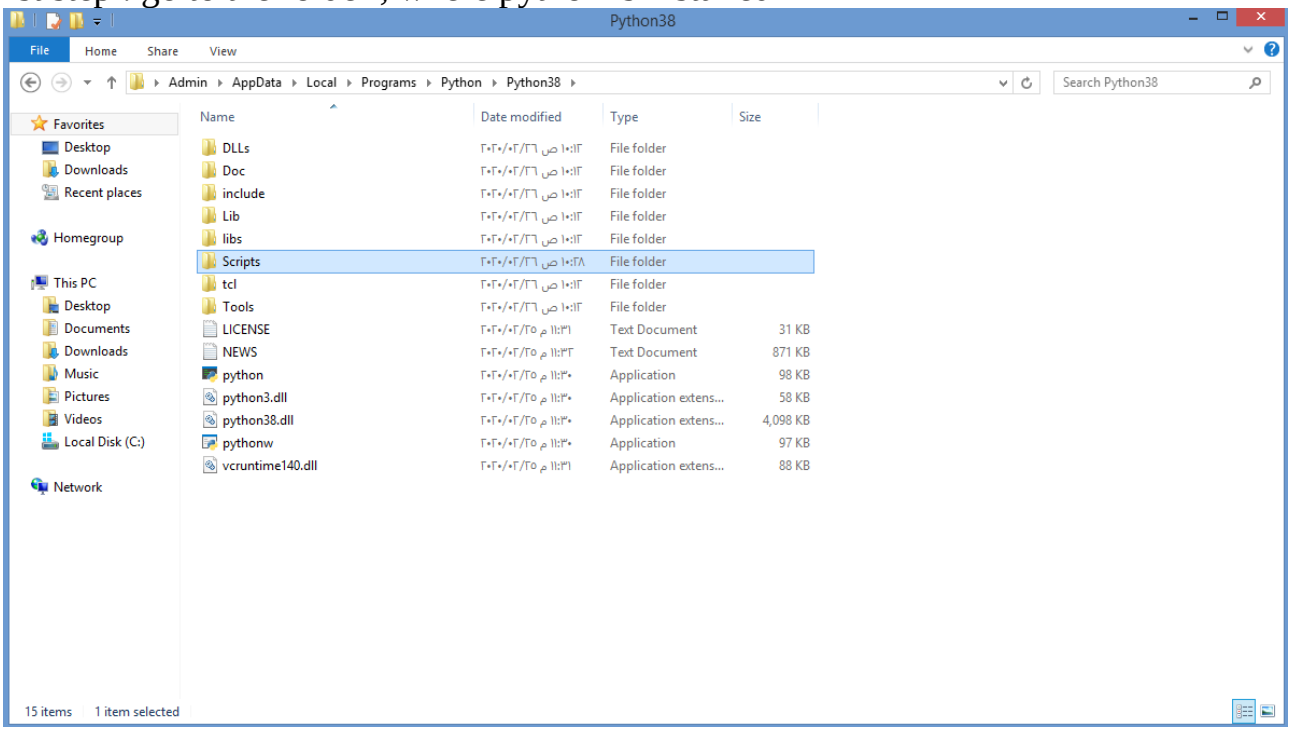
Example 5 : from start (50,250) to goal (53,820) the path outs as shown



Finally : to make the "mycode.py" as shown in the previous image works fine , the required libraries must be there in the python env. , and they are :

```
import numpy as np
import queue
import time
import sys
import os
import collections
from matplotlib import cm
from PIL import Image
```

how to install missing python libraries on windows
1st step : go to the folder , where python is installed



then : copy the address to the folder "Scripts"
then use cmd , and change it`s dir to that folder including "Scripts" folder
then the command line must be as shown in image below,
then to install library type "pip install {library name"
for example : for "numpy"
type : "pip install numpy"
or , for "matplotlib"
type : "pip install matplotlib" or a missing library