Data Structures – Assignment 8
IDC, Spring 2022

Lecturer: Prof. Yael Moses, Dr. Ben Lee Volk

TAs: Yael Hitron, Guy Kornowski, Elad Tzalik

Submission day: 1.6.22 – (you can use an extension and submit by 4.6.22).

Guidelines:

- You can submit this assignment in pairs (no triplets)
- In order to ask a question regarding the assignment on the piazza, the title of the question must be of the form 'assignment x', and it should be written in English only.
- Do not import or use any libraries, except the ones provided.

Honor code:

- Do not copy the answers from any source.
- You may work in small groups but write your own solution - write whom you worked with.
- Cheating students will face Committee on Discipline (COD).
- Do not forget – you are here to learn!

The Assignment

In this assignment you will write a Java implementation of the Dynamic Disjoint Sets ADT (Union-Find), and then use it to verify whether a given maze is solvable. We next describe each one of the classes we will use.

UnionFind (Disjoint Sets)

Open the supplied Java class *UnionFind*, it contains the skeleton for the union-find data structure, using up-trees. Fill in the following public methods:

- *public UnionFind(int numElements);* The constructor initializes the data structure with numElements sets, each containing a single element. The elements are numbered sequentially from 1 to numElements.

- *public void union(int i, int j);* Unites the sets that contain i and j, or does nothing if they are already in the same set. Note that i and j must be the representatives of their sets. The method should use the weighted union methodology. That is, a tree with less nodes should be put beneath a tree with more nodes.

- *public int find(int i);* Returns the representative of the set that contains i, and applied path compression to the traversed path.

- *public int getNumSets();* Returns the current number of sets.

The supplied main() method performs several tests on the UnionFind class, you may use them to test your class. Note that this will only give you an indication if you are on the right direction. Passing those tests is not a guarantee that your implementation is correct. You should write more tests of your own. All of your functions should be implemented to be as efficient as possible, as seen in class.

Maze

Open the supplied Java class *Maze*. The class is supposed to read an image representing a maze, where the color of the background (either black or white) is given as a parameter. The start and end points of the image are given as red pixels – you can assume that the image contains all black or white pixels, except for exactly two pixels that mark the beginning and end of the maze. The maze should be decomposed into its connected components. A connected component is a set of pixels with the same color, such that between any two pixels there exists a path through neighboring pixels that are contained in the set. You will need to fill in the following public methods:

- *public maze (String fileName, Color c);* The constructor accepts the name of a file containing a .jpg or .png image, and the color of the background. It reads the image using the provided DisplayImage class (see below), and then creates an instance of UnionFind (see above), and uses it to decompose the image into connected components. In order to find the start/end points of the maze (i.e., the red pixels), you can use the *isRed()* method provided in the DisplayImage class (detailed below). After finding the start/end points, you should save their coordinates (as fields of the class), and then color the two red pixels with the given background color so that the decomposition process can include them in the correct connected components.

- *public void connect (int x1, int y1, int x2, int y2);* Checks that pixel (x1, y1) and pixel (x2, y2) both belong to the same image area (have the same color), and if they are, connects the components that they belong to (unless they are already part of the same component). You may assume that (x1, y1) and (x2, y2) are neighboring pixels (as detailed in the implementation details below).

- *public boolean areConnected (int x1, int y1, int x2, int y2);* Checks whether or not the two given pixels belong to the same component.

- *public int getNumComponents();* Returns the current number of components.
- *public boolean mazeHasSolution();* Returns true if and only if the maze has a solution. That is, if the start and end points of the maze belong to the same connected component.

DisplayImage

For your convenience, a class called DisplayImage is provided to you, to support basic image read/display/manipulation operations. It supports the following methods:

- *public DisplayImage (String filename);* The constructor creates a new DisplayImage instance from a given image file.
- *public void show();* Displays the image in a new window.
- *public int height(); public int width();* Return the height/width of the image.
- *public Color get (int x, int y);* Returns the color of pixel (x,y).
- *public void set (int x, int y, Color c);* Sets the color of pixel (x,y) to be *c*.
- *public boolean isOn (int x, int y);* Checks whether a pixel is black or white. It will return true if the pixel has an intensity below 128 (black), and false otherwise.
- *public boolean isRed (int x, int y);* Checks whether a pixel is red or not (it does so by comparing the red component of the RGB colors to the green and blue components, to see if it is significantly larger).
- *public void save (String filename);* The save() method saves the current image into a new file with the given name.

Implementation Details

- Each pixel in the input images has some intensity between 0 and 255, but the specific color is irrelevant. Furthermore, we treat them as binary images, where each pixel is either on or off. The method isOn() in DisplayImage will return true if the pixel has an intensity below 128, and false otherwise. The only exception to this is the two red pixels. Those two pixels represent the start and end points of the maze.
- To decompose an image into connected components, you can view the image as a graph, where each pixel is a node. For any two neighboring pixels, if they both belong to the background or both belong to the maze itself (i.e., both have the same intensity), then there will be an edge connecting them. Finding connected components in this graph is equivalent to visually segmenting the image.
- We use 4-connectivity, so each pixel has 4 neighbors: left/right or above/below.

- The constructor of the Maze class should traverse the image's pixels, and connect each pixel with its neighbors. An efficient implementation will process each such edge (pair of neighboring pixels) **exactly once.** While traversing the image, the constructor should determine the start and end points of the maze and color them with the given background color.

- After calling the constructor with a given image file and color, the mazeHasSolution() function should return the correct solution.

- Each pixel has two coordinates; However when we add elements to a Union-Find data structure we need to provide a *single* integer as a key. To generate unique keys from (x, y) coordinates, we can use the formula: $y \cdot width + x$. This will create, for each pixel, a unique id between 0 and $width \cdot height - 1$.

- You can test your code on the provided images. Each pair of images contains one version of the maze that can be solved (mazes number 1,3,5,7), and one that cannot (mazes 2,4,6,8). The mazes with filenames that starts with 'black' have black background, and the rest of the mazes have white background.

- The supplied function *getComponentImage()* will create a visual display of the different connected components. Once you finish writing the class, you may use it to test your solution. While testing your solution use the static variables Color.white and Color.black in the maze constructor.

Examples. Consider the images given in Figure 1, which are two of the images provided as part of the exercise package. In each image, the background is white, and there are two specific pixels colored red – these mark the start and end of the maze (no significance to which is which). After finding them and saving their position, you should color them white, and then run the segmentation, and use it to check whether a path between the two end points exists or not. For the left image, the answer is yes, and for the right image the answer is no.

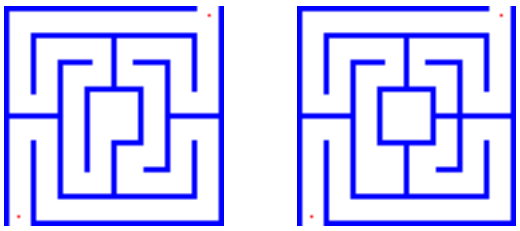

Figure 1: Input examples. The left maze is solvable, and the right maze is not solvable.

General Guidelines. All methods should perform as efficiently as possible.
You may add classes and methods that were not defined in the given API, as long as they conform to proper Object Oriented Design.
You are responsible for testing your code.
You should provide full documentation of all classes and methods.
As long as the provided input is legal, your program should not crash. Make sure you consider the possible edge cases.

Submission:

You may submit the assignment in pairs. This is not mandatory but recommended. You are **not** allowed to submit in groups of three of any number $k > 2$.

Before submitting this assignment, take some time to inspect your code and check that your functions are short and precise. If you find some repeated code, consider making it into a function. Make sure your code is presentable and is written in a good format. Any deviations from these guidelines will result in a point penalty.
Make sure your code does not suffer from compilation errors. **Code which does not compile will not be graded**.

Submit a zip file with the following files only:

- Maze.java
- UnionFind.java

The name of the zip file must be in the following format "ID-NAME.zip", where "ID" is your id and "NAME" is your full name. For example, "03545116-Allen_Poe.zip". If you submit as a pair, the zip file should be named in the following format "ID-NAME-ID-NAME.zip". For example, "03545116-Allen_Poe-02238761-Paul_Dib.zip".