Data Structures – Assignment 8
IDC, Spring 2018

Submission day: 12.6.2018– (you can use an extension and submit by 15.6.2018)
Honor code:
1. Do not copy the answers from any source
2. You may work in small groups but write your own.
3. Cheating students will face Committee on Discipline.

The Assignment

In this assignment you will write a Java implementation of the Dynamic Disjoint Sets ADT (Union-Find), and then use it to test whether a maze given as an input image can be solved or not.

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);`
- `public void union (int i, int j);`
- `public int find (int i);`
- `public int getNumSets();`

The constructor initializes the ADT with `numElements` sets, each containing a single element. The elements are numbered sequentially from `1` to `numElements`.

The `union()` method 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. A tree with less nodes should be put beneath a tree with more nodes.

The `find()` method returns the representative of the set that contains `i`, and applied *path compression* to the traversed path.

The `getNumSets()` method 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 divided into foreground and background (dark or light pixels, respectively), representing a maze. The maze should be decomposed into its connected components. A connected component is a set of pixels that all belong either to the background or to the foreground, such that between any two pixels there exists a path through neighboring pixels in the set:

- `public maze (String fileName);`
- `public void connect (int x1, int y1, int x2, int y2);`

-
  ```
  public boolean areConnected (int x1, int y1,
                               int x2, int y2);
  ```
- `public int getNumComponents();`
- `public boolean mazeHasSolution();`

The constructor accepts the name of a file containing a `.jpg` or `.png` image. 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.

The `connect()` method checks that pixel `(x1, y1)` and pixel `(x2, y2)` both belong to the same image area (foreground or background), 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 below).

The `areConnected()` method checks whether or not the two given pixels belong to the same component.

The `getNumComponents()` method returns the current number of components.

The `mazeHasSolution()`method 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.

You should also add tests to the `main()` method which performs tests on the `Maze` class, but this should not be submitted as part of your solution.

Maze Solution

Finally, you will use the results of the image processing, to determine whether the given maze has a solution or not. 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. You can use the `isRed()` method provided in the `DisplayImage` class to check whether a given pixel is red or not. After finding the start/end points, you should save their coordinates (as fields of the class), and then color the two red pixels white, so that the decomposition process can include them in the correct connected component.

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 (we assume that we have a dark image on a light background, therefore a high intensity means that the image is off).

  The only exception to this is the two red pixels. Those two pixels represent the start and end points of the maze. These should be colored white before the decomposition begins, because we want them to be treated as part of the background. Coloring the pixels white can be done using:

  ```
  image.set (x, y, new Color (0xFF, 0xFF, 0xFF)) ;
  ```

- 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 foreground, or they both belong to the background, 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, not 8 (only the ones to the left/right or above/below are considered, no diagonal neighbors).

- 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.
- Also, as noted above. While traversing the image, the constructor should determine the start and end points of the maze and color them white.
- After calling the constructor with a given image file, the `mazeHasSolution()` function should return the correct solution.
- Each pixel has two coordinates; However when we add elements to a Union-Find ADT we need to provide a single integer as a key. To generate unique keys from (x, y) coordinates, we can use the formula: `y * width + x`. This will create, for each pixel, a unique id between 0 and `width * height - 1`.
- You can test your code on the provided images: `maze1-8.png`. 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 supplied `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.

Image Manipulation

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);`
- `public void show();`
- `public int height();`
- `public int width();`
- `public Color get (int x, int y);`
- `public void set (int x, int y, Color c);`
- `public boolean isOn (int x, int y);`
- `public boolean isRed (int x, int y);`
- `public void save (String filename);`

The constructor creates a new `DisplayImage` instance from a given image file.

The `show()` method displays the image in a new window.

The `height()` and `width()` methods return the size of the image.

The `get()` and `set()` methods support querying and manipulating individual pixel values.

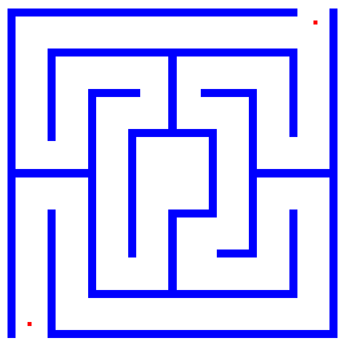
The `isOn()` method checks whether a pixel is foreground (black) or background (white), and can thus be used for segmentation. It will return `true` if the pixel has an intensity below 128, and `false` otherwise.

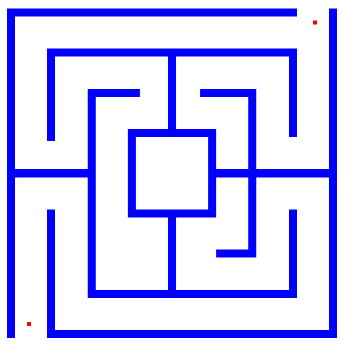
The `isRed()` method checks whether a pixel is red or not (it does so by comparing the red component of the RGB colors to the green and blue components, to see if it is significantly larger).

The `save()` method saves the current image into a new file with the given name.

Examples

Consider the following images, which are two of the 8 images provided to you as part of the exercise package:


`maze5.png`


`maze6.png`

In each image, 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 `maze5.png`, the answer is yes, for `maze6.png` it is no.

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.
- Make sure your code is presentable and is written in good format. Any deviations from these guidelines will result in a point penalty.
- Make sure your code can be compiled. **Code which does not compile will not be graded.**
- Submit a zip file with the following file 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.
- 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".