Data Structures
Homework 8

Subjects:
UnionFind

Honor code:

1. Do not copy the answers from any source
2. You may work in small groups but write your own solution (mention this in the homework)
3. Cheating students will face Committee on Discipline (COD)

Submission date - 14/6 - through moodle

The Assignment

In this assignment you will write a Java implementation of the UnionFind ADT, and then use it to test whether a maze given as an input image can be solved or not.

Union Find

Write a Java class called `UnionFind` that provides the Union-Find functionality, using up-trees. The class should contain 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(i,j)` method unites the sets that contain `i` and `j`. Note that `i` and `j` must be the **representatives(leaders)** of their sets. The method should use the ***weighted union*** methodology.

The `find(i)` 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.

Segmentation

Write a Java class called `Segmentation` that reads an image divided into foreground and background (dark or light pixels, respectively), and segments the pixels into 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 Segmentation (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();`

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 segment 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).

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.

Maze Solution

Finally, you will use the results of the image segmentation, 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, and then color the two red pixels white, so that the segmentation process can include them in the correct segment.**

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. These should be colored white before the segmentation runs, 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 segment 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).
- When traversing the image pixels, we connect each pixel with its neighbors. An efficient implementation will process each such edge (pair of neighboring pixels) exactly once.
- 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 + 1`. This will create, for each pixel, a unique id between 0 and `width * height`.
- To determine whether the given maze is solvable or not, you should write a method of the `Segmentation` class with the following signature:

  ```
  public boolean mazeHasSolution();
  ```
- 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).

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.

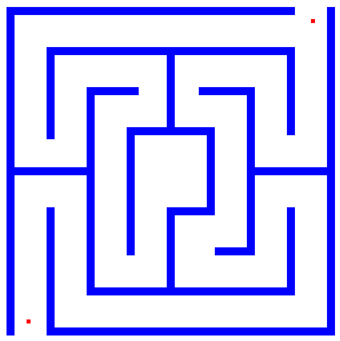
The `isOn()` method checks whether a pixel is foreground (black) or background (white), and can thus be used for segmentation. It will return `true` if the pixel has an intensity below 128, and `false` otherwise.

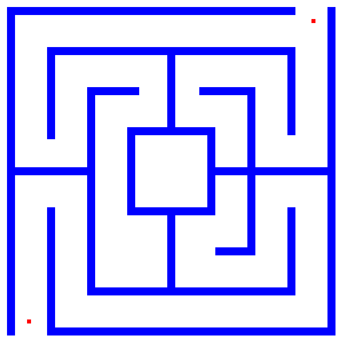
The `isRed()` method checks whether a pixel is red or not (it does so by comparing the red component of the RGB colors to the green and blue components, to see if it is significantly larger).

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.

Testing

You are provided with a basic testing pack - SegmentationTest.java.

You are also provided with an SegmentationTestOutput.txt for the given set of mazes.

Compare your output with the expexted one. Also the provided tests are extremely basic - test your code more carefully.

Submission instructions:

1. The submission is in pairs (not obligatory but highly recommended).
2. If you submit in pari the submission file will be ID_ID.zip (e.g. 123456789_987654321.zip). If you submit by yourself the file will be named ID.zip (e.g. 123456789.zip)
3. Don’t create any directories in the **zip** - it should include just the java files of the solution
4. All the classes must belong to the default package (that is - no package)
5. If your code won’t compile - it won’t be checked
6. In the check process we will unzip your file and replace the SegmentationTest.java with another set of tests.

Grading

● Correctness:	50%
● Efficiency:	10%
● Robustness:	10%
● Architecture & Design:	10%
● Documentation & Readability:	10%
● Submission instructions	10%