Hole Filling

Abstract

The goal is to build a small image processing library that fills holes in images, along with a small command line utility that uses that library, and answer a few questions. The coding tasks must be implemented using one of the following languages: C, C++, Objective-C, Swift or Java. If you're not feeling proficient in any of these languages, please let us know in advance. Make sure to pay close attention to the design of your code, and not just to the completeness or efficiency of the code.

The library must support filling holes in grayscale images, where each pixel value is a `float` in the range `[0, 1]`, and hole (missing) values which are marked with the value `-1`.

The library should provide the ability to fill a hole in an image, based on the following algorithm:

Hole Filling Algorithm

Definitions:

- I : the input image.
- $I(v)$: color of the pixel at coordinate $v \in \mathbb{Z}^2$.
- B : set of all the boundary pixel coordinates. A boundary pixel is defined as pixel that is connected to a hole pixel, but is not in the hole itself. Pixels can be either 4- or 8-connected. See this for more info.
- H : set of all the hole (missing) pixel coordinates. You can assume the hole pixels are either 4- or 8-connected.
- $w(v, u)$: weighting function which assigns a non-negative float weight to a pair of two pixel coordinates in the image.

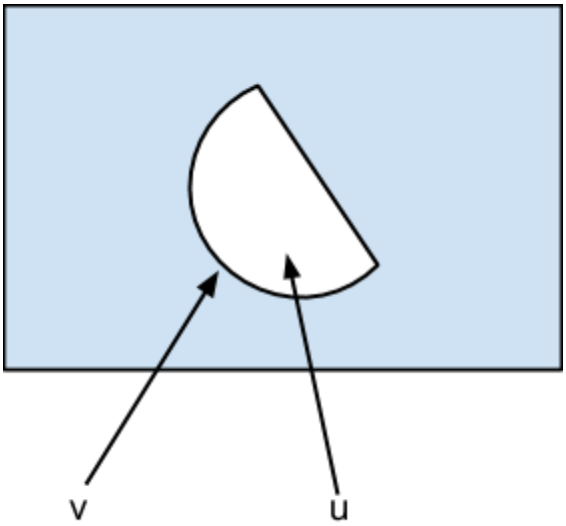

Figure 1: an image with a hole

Algorithm:

Find the boundary B and the hole H in a given image. For each $u \in H$, set its value to:

$$I(u) = \frac{\sum_{v \in B} w(u,v) \cdot I(v)}{\sum_{v \in B} w(u,v)}$$

Requirements

1. The default weighting function for the algorithm is $w_{z,\epsilon}(u,v) = \frac{1}{\|u-v\|^z + \epsilon}$, where ε is a small float value used to avoid division by zero, and $\|u-v\|$ denotes the euclidean distance between u and v. The values z, ε should be configurable.
2. The library needs to support any arbitrary weighting function.
3. The library needs to support both the 4- and the 8-connectivity options.
4. Write a command line utility that accepts an input image file, z, ε and connectivity type, fills the hole and writes the result to an image file.
    a. The library should work with grayscale images, as noted above, with -1 marking a missing color. But the command line utility may accept (and write to) other image formats, and perform the required conversions. This might be more convenient (and easier) to implement, because the most popular image formats do not support `-1` values. You are free to choose what format (or formats) the command line utility accepts, and how to describe a hole in an image. Our suggestion is to accept 2 input RGB images, one for the image and one for a mask that defines the hole, and then merge the two images by converting RGB to grayscale and

applying the mask to carve out the hole.
    b. Note that the command line utility needs not support an arbitrarily weight function, but only the default one.
5. For loading and saving the image, and for conversion between different image formats, you can use OpenCV or any other similar library. You are allowed to use the library's data structures to hold the image, but you're not allowed to use any algorithmic functionality (related to image processing) provided by it.
6. There is no need to create (build) the library separately from the command line utility, but the "library" code needs to be easily extractable to an actual library should you want to build one.
7. For simplicity, you can assume that every image has only a single hole.
8. There's no need to handle the corner case of holes that touch the boundaries of the image.

Questions

Please answer the following questions and add them to a separate file in your submission.

1. If there are m boundary pixels and n pixels inside the hole, what's the complexity of the algorithm that fills the hole, assuming that the hole and boundary were already found? Try to also express the complexity only in terms of n.
2. Describe an algorithm that approximates the result in $O(n)$ to a high degree of accuracy. As a bonus, implement the suggested algorithm in your library in addition to the algorithm described above.
3. Bonus (hard!): Describe and implement an algorithm that finds the *exact* solution in $O(nlogn)$. In this section, feel free to use any algorithmic functionality provided by external libraries as needed.

Good luck!