HW4-110810006林君曆

Watershed Segmentation(Meyer's flooding algorithm)

1-1. Mark the area you want to segment.

The `interactive_segmentation` function allows for interactive marking of an image using mouse input and keyboard event handling. It loads an image from a specified path and displays it in a window where users can draw segments with different colors.

Operation

1. Mark(clicking and dragging the mouse)
2. Next mark(label+1)(Pressing 'n')
3. Save(Pressing 's')
4. Exit(Pressing 'q')

Return a 2d array of labels(0 = unmarked, 1 = label 1, 2 = label 2 ...)

1-2. Region growing

This part includes two functions, `sobel_filters` and `apply_watershed`

1. **apply_watershed**:
   a. Compute gradient magnitude using Sobel operator as the priority queue criteria
   b. Populate the priority queue with neighbors of marked pixels and change their label to -2(means it is in queue)
   c. Keep popping out pixel from the queue
      - If the neighbors are all in a same label, then assign the label to the pixel.
      - Otherwise, mark the pixel -1(means edge)
      - Add its unmarked neighbors to the queue
   d. Repeat (c) until no pixels in the queue, which means all the points are marked
2. **sobel_filters**:

> 💡 The gradient magnitude of an image measures the rate of change in intensity at each pixel position. So, we can start growing the region from the smallest variation pixel.

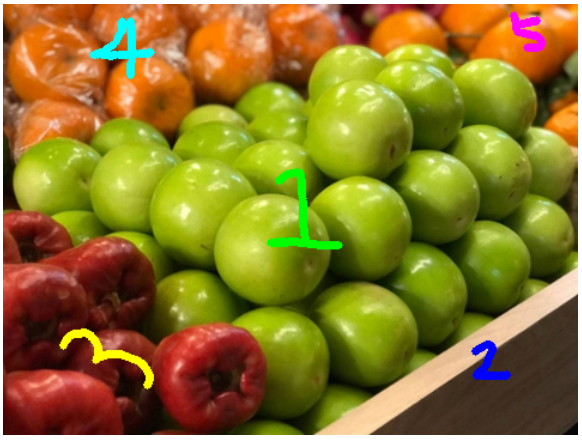
- This function calculates the gradient magnitude of an image using the Sobel operator, a popular method for edge detection. Define two Sobel kernels, `Gx` and

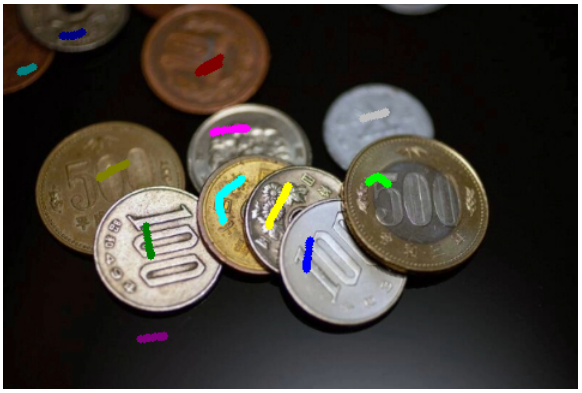
`Gy`, to detect horizontal and vertical edges respectively.

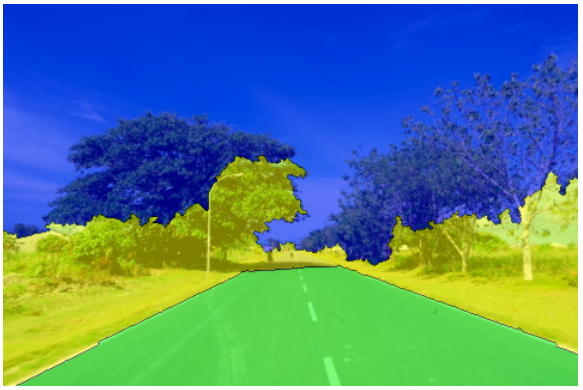
- The image is padded to handle border effects during convolution.
- Apply these kernels to the image to compute the gradients in the x and y directions.
- The overall gradient magnitude at each pixel is calculated as the Euclidean norm of the x and y gradients.
- Lastly, this gradient magnitude is then normalized to the range of 0-255 and returned along with the individual x and y gradients.

Result images

After making some different marks, I found that this algorithm tends to separate two regions at locations where there is a significant change in pixel values, typically where shadows are present. The second picture's result is better, because of the clear edges.







Additional efforts

When I tried using gradient magnitude as the criteria, the results were bad for reasons I couldn't identify (perhaps I made some errors). Consequently, I searched for alternative methods online and came across a paper [1] and an explanation [2]. However, my code remained incomplete, and I was unable to finish it. This algorithm is very similar to Meyer's flooding algorithm. The priority criteria in this method is the difference of gray level between a point and the average gray level of its neighboring region. For this, I created a dictionary to keep track of each region's information (mean and pixel count). The rest of the algorithm is quite similar to the Meyer's one: (1) initialize the priority queue by inserting the neighbors of the seed points, (2) pop the point with the highest priority and add it to a neighboring region (updating the region's count and mean), (3) then insert the unmarked neighbors of that point into the queue. This cycle continues until all points are marked.

However, the final results were not as good as expected. I suspect the issue was that the priority for points added to the priority queue was calculated using the current regional mean, but the mean of the region adjacent to points already in the queue might have been updated afterwards, making the priority of those points outdated. Therefore, a dynamic update of the priorities would be necessary. However, when I tried to implement this, the need to re-heapify the queue took nlogn time and, given there are m*n points, this made the computational complexity too high. Consequently, I did not complete this approach and reverted to using the original gradient magnitude as the criteria.

Main function code is showing below

```python
class RegionInfo:
    def __init__(self, count=0, mean=0): # empty constuctor
        self.count = count
        self.mean = mean # the mean of the gray_level values of every points
    def __init__(self, gray_levels): # construc by a set of seeds in a region
        self.count = len(gray_levels)
        self.mean = np.mean(gray_levels)
    def addPoint(self, gray_level):
        self.mean = (self.mean*self.count + gray_level)/(self.count+1)
        self.count += 1
    def addPoints(self, gray_levels):
        self.mean = (self.mean*self.count + np.sum(gray_levels)) \
                                    /(self.count+len(gray_levels))
        self.count += len(gray_levels)

def region_filling(img_gray_level, img_labels):
    rows, cols = img_labels.shape
    priority_queue = []
    d = dict() # mark:[graylevels]
    # collect the gray_level of each mark
    for i in range(rows):
        for j in range(cols):
            if img_labels[i, j] > 0:  # If pixel is initially marked
                if img_labels[i, j] in d:
                    d[img_labels[i, j]].append(img_gray_level[i , j])
                else:
                    d[img_labels[i, j]] = [img_gray_level[i , j]]
    # transfer d[mark:[gray_levels]] to d[mark:RegionInfo]
    for k, v in d.items():
        d[k] = RegionInfo(v)
    # put neighbors of seeds to priority queue
    # with the diff(pixel_intensity, region_mean)
    for i in range(rows):
        for j in range(cols):
            mark = img_labels[i, j]
            if mark > 0:  # If pixel is initially marked
                for di, dj in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                    ni, nj = i + di, j + dj
                    if 0 <= ni < rows and 0 <= nj < cols and img_labels[ni, nj] == 0:
                        region_mean = d[mark].mean
                        gray_level = img_gray_level[ni, nj]
                        heapq.heappush(priority_queue, (abs(region_mean-gray_level), ni, nj))
                        img_labels[ni, nj] = -2  # Mark as in queue
```

```python
    # Process the queue
    while priority_queue:
        _, x, y = heapq.heappop(priority_queue)
        i += 1
        neighbors = [img_labels[x + dx, y + dy] for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]
                     if 0 <= x + dx < rows and 0 <= y + dy < cols]
        unique_labels = set(filter(lambda l: l > 0, neighbors))
        if len(unique_labels) == 1:
            img_labels[x, y] = unique_labels.pop()
            m = max(img_labels[x, y], m)
            d[img_labels[x, y]].addPoint(img_gray_level[x, y])
        else:
            img_labels[x, y] = -1  # Mark as edge

        # Add unmarked neighbors to the queue
        mark = img_labels[x, y]
        if mark > 0:
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nx, ny = x + dx, y + dy
                if 0 <= nx < rows and 0 <= ny < cols and img_labels[nx, ny] == 0:
                    region_mean = d[mark].mean
                    gray_level = img_gray_level[nx, ny]
                    heapq.heappush(priority_queue, (abs(region_mean-gray_level), nx, ny))
                    img_labels[nx, ny] = -2  # Mark as in queue

    return img_labels
```

Terrible Results :(

Reference

[1] R. Adams and L. Bischof, "Seeded region growing," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 16, no. 6, pp. 641-647, June 1994, doi: 10.1109/34.295913.

https://ieeexplore.ieee.org/document/295913/authors#authors

[2] Explanation of Seeded region growing algorithm

https://blog.csdn.net/fanqiliang630/article/details/111464190