Code Documentation

Challenge

The scenario is that you are working with microbiologist researchers who are investigating cancer in parasitic microorganisms. The microbiologists have access to an electron microscope that is capable of capturing very high resolution images of the microorganisms; each image that is captured by the microscope is of a single microorganism, which shows up in the image as a single blob of arbitrary shape (the microscope has zoomed in on each parasite so that the parasite occupies 25% or more of the total area of the image). For the purposes of this challenge, you can assume that each pixel of the image is either black if it is part of the blob, or white if it is the space surrounding the blob. The images are 100,000x100,000 pixels each, and many thousands of such images will be captured (one for each parasite in a colony).

Additionally, the researchers have injected each of the parasites with a luminescent dye that permeates their body and lights up different parts of it (imagine the dye flowing through blood vessels in the parasite). There is a separate sensor that captures high resolution images (also 100,000x100,000 pixels) that show where the dye is lit and where it isn't. Unfortunately, there is some leakage of the dye to the space surrounding each parasite, so the dye sensor sometimes also picks up the presence of dye outside of the parasite.

Note that the dye sensor and the microscope both have the same resolution, and capture their images at the same instant in time, but produce different images; the microscope images only show the parasite's whole body, while the images produced by the dye sensor only show where there is dye (even if it is outside of the parasite's body).

A parasite is deemed to have cancer if the total amount of dye detected in its body exceeds 10% of the area occupied by the parasite in the image.

The researchers would like to store images of all the parasites that they have looked at, and in addition they would like to store images of the parasites that show where the dye is lit up, but only for those parasites that have cancer. It is expected that fewer than 0.1% of the parasites will have cancer.

The researchers are looking for you to help them efficiently store and process all the data that they will be generating.

Questions

1. Come up with efficient data structures to represent both types of images: those generated by the microscope, and those generated by the dye sensor. These need not have the same representation; the only requirement is that they be compact and take as little storage space as possible. Explain why you picked the representation you did for each image type, and if possible estimate how much storage would be taken by the images. What is the worst-case storage size in bytes for each image representation you chose?
2. Before the researchers give you real images to work with, you would like to test out any code you write. To this end, you would like to create "fake" simulated images and pretend they were captured by the microscope and the dye sensor. Using the data structures you chose in (1) above, write code to create such simulated images. Try and be as realistic in the generated images as possible.
3. Using the simulated images generated by the code you wrote for (2) above as input, write a function to compute whether a parasite has cancer or not.
4. You give your code from (3) to the researchers, who run it and find that it is running too slowly for their liking. What can you do to improve the execution speed? Write the code to implement the fastest possible version you can think of for the function in (3).
5. What other compression techniques can you suggest for both types of images (parasite and dye)? How would they impact runtime? Can you compute actual runtime and storage costs for typical images (not oversimplified image such as a circle for the parasite, or simple straight lines or random points for dye) in your code? The measurements should be done on your computer with an actual image size of 100,000x100,000 pixels (and not a scaled down version).
6. Describes what tools you used to solve the challenge, particularly any LLM techniques.

Generating Microscope Images

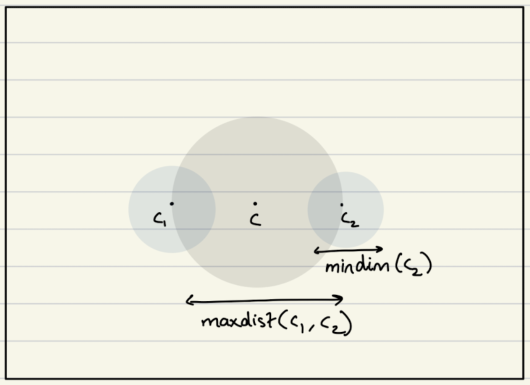
- Defined a function that generates microscope images of parasites given the dimensions of the image.

```
def generate_microscope_image(width, height):
```

- Defined variable `target_black_area` as the task specified that the *"parasite occupies 25% or more of the total area of the image"*
- Also defind total_black_area which represents the area occupied by the parasite in the generated image. It is initalised to 0, but is calculated by computing the number of pixels with value `255`. This is because I am generating images with pixels taking values either `0 or 255`. 255 represents the existence of a parasite

```
    center = (width // 2, height // 2)
image_area = width * height

target_black_area = 0.25 * image_area
total_black_area = 0

composite_image = np.zeros((height, width), dtype=np.uint8)
total_black_area = np.count_nonzero(composite_image == 255)
count = 0
```

- The image generation task is completed when the `total_black_area` reaches `target_black_area` (i.e when 25% of the image is occupied by a parasite)
- To generate the parasite, a random ellipse is generated (with random axis lengths and angle of rotation). The center of the ellipse is computed by adding noise to the center of the image. This iteratively takes place until we reach the desired parasite area. This overlaying of random ellipses mimics the parasite body
- The reason for choosing this algorithm and the parameters in the algorithm is:
    - The parasite should be a single blob
    - There should not be no white space conceded in the body of the blob
- Derivation of parameters:
    - Even in the worst case, this inequality does not hold.



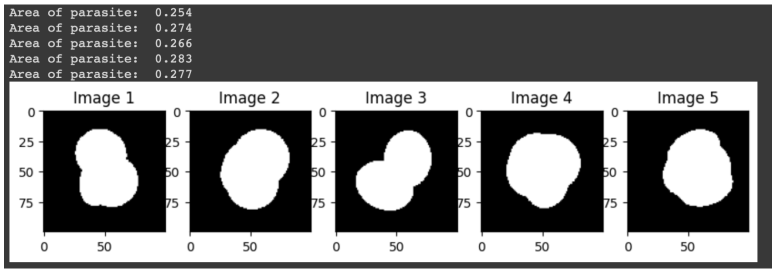
there will be whitespace in the blob if:

$$\underset{\underset{\displaystyle 2 * \frac{size}{12}}{\Downarrow}}{maxdist(c_1, c_2)} > \frac{mindim(c_1)}{2} + \underset{\underset{\displaystyle 2 * \frac{size}{12}}{\Downarrow}}{\frac{mindim(c_2)}{2}}$$

```
while total_black_area < target_black_area and count<10:
    count += 1
    # Random parameters for ellipse
    axis_x = np.random.randint(min(width, height) // 6, min
    axis_y = np.random.randint(min(width, height) // 6, min
    angle = np.random.randint(0, 180)  # Random angle of rot

    # Perturb center coordinates with noise
    noise_x = np.random.randint(-min(width, height) // 12, r
    noise_y = np.random.randint(-min(width, height) // 12, r
    perturbed_center = (center[0] + noise_x, center[1] + noi



    shape = np.zeros((height, width), dtype=np.uint8)
    cv2.ellipse(shape, perturbed_center, (axis_x, axis_y), a

    # Add distorted shape to list
    composite_image = np.maximum(composite_image, shape)
    total_black_area = np.count_nonzero(composite_image == 2
print("Area of parasite: ", "{:.3f}".format(total_black_area
return composite_image
```

Output:

5 images were created with dim 100 x 100


```
Area of parasite:  0.254
Area of parasite:  0.274
Area of parasite:  0.266
Area of parasite:  0.283
Area of parasite:  0.277
```

These images were then enlarged to 10000 x 10000

```
#Enlarging images
enlarged_microscope_images = []
for image in sample_microscope_images:
    enlarged_image = np.repeat(np.repeat(image, 100, axis=0), 10
    enlarged_microscope_images.append(enlarged_image)


enlarged_microscope_images[0].shape
```

Other approaches:

- Random Walk
- Perlin Noise

Generating Dye sensor images

- Step (1): Mimic the point of injection of dye on the parasite:

    - Randomly choose coordinates in the center region of the the image. If this point is on the parasite, return it. Else continue

```
def find_point_of_injection(self):
    injection_val = 0
     while True:
      injection_x = np.random.randint(self.size // 3, 2*self.si
      injection_y = np.random.randint(self.size //3, 2*self.size
      injection_val = self.image[injection_x][injection_y]
      if injection_val == 255:
           self.microbe_dyed_pixel_count += 1
           self.dyed_pixels.add(((injection_x, injection_y)))
           self.outer_pixels.add(((injection_x, injection_y)))
           return (injection_x, injection_y)
```

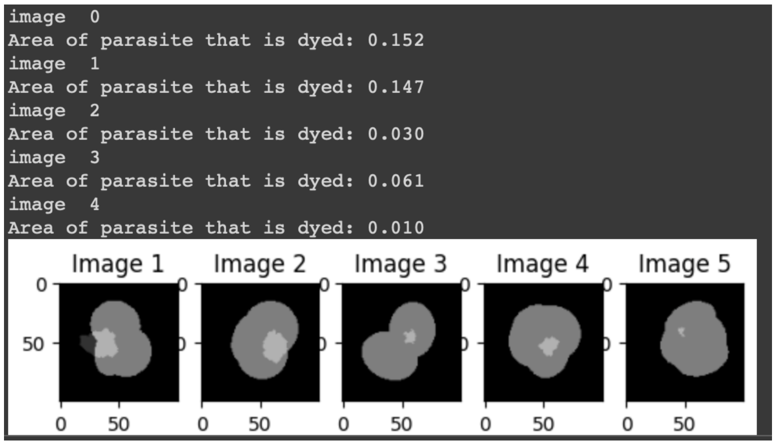
- Step(2): Mimic the flow of dye in blood vessels:
    - Pick a point that is already dyed and grow it in a random direction

```
    def random_update(self):
        point = random.choice(list(self.dyed_pixels))
        #right, left, up, down
        neighbors = [(point[0]+1, point[1]), (point[0]-1, point
        #print("Neigbors: ", neighbors)
        neighbors = [(x, y) for x, y in neighbors if 0 <= x < se
        #print("Neigbors: ", neighbors)
        if len(neighbors) == 0:
            #print("No neighbors")
            return
```

- Step (3): Mimic fluid continuity
    - Take all the outer pixels of dye and grow the dye to its neighbors

```
    def grow(self):
        grow_pixels = list(self.outer_pixels)
        self.outer_pixels = set()
        for pixel in grow_pixels:
            neighbors = [(pixel[0]+1, pixel[1]), (pixel[0]-1, pi
            neighbors = [(x, y) for x, y in neighbors if 0 <= x

            valid_neighbors = [neighbor for neighbor in neighbor
            for neighbor in neighbors:
                if neighbor in valid_neighbors and neighbor not
                    self.microbe_dyed_pixel_count += 1
                self.dyed_pixels.add(neighbor)
                self.outer_pixels.add(neighbor)
```

Output:



```
image  0
Area of parasite that is dyed: 0.152
image  1
Area of parasite that is dyed: 0.147
image  2
Area of parasite that is dyed: 0.030
image  3
Area of parasite that is dyed: 0.061
image  4
Area of parasite that is dyed: 0.010
```

The above approach mimics fluid flow through an organism and it pertains to the task as some of the dye can overflow from the microbe. Similar to the previous image generation task, smaller scale images were generated and then enlarged.

Storing and Compressing the images

- Step (1): Compute the bounding box
    - **bounding_box_coordinates:** ((x1,y1), (x2,y2)) These coordinates represent the pixel locations of the top left and bottom right corners of the bounding box. The bounding

box is the smallest rectangle that contains every parasite/dye pixel.

```
def compute_bounding_box(self, image, val):
        rows, cols = image.shape

        top_row = next((i for i in range(rows) if val in image[
        if top_row is None:
            return None

        bottom_row = next((i for i in range(rows - 1, -1, -1) i
        if bottom_row is None:
            return None

        left_col = next((j for j in range(cols) if val in image
        if left_col is None:
            return None

        right_col = next((j for j in range(cols - 1, -1, -1) if
        if right_col is None:
            return None

        self.bounding_box = ((top_row, left_col), (bottom_row,
        return ((top_row, left_col), (bottom_row, right_col))
```

- Step (2): Compute compressed RLE Encoding
    - Compute the first pixel value (top-left) in the bounding box
    - Compute lengths of consecutive runs
    - Compute the number of pixels that represent parasite/dye (helpful for computing area for cancer prediction)

```
    def compute_rle_encoding(self, image, val):
        if self.bounding_box is None:
            return None, None  # No bounding box, return None
        top_left, bottom_right = self.bounding_box
        min_row, min_col = top_left
        max_row, max_col = bottom_right

        rle_encoding = []
        start_value = image[min_row][min_col]  # First value in
        current_color = start_value
        run_length = 0
        pixel_count = 0  # Initialize count of pixels with the s
        for row in range(min_row, max_row + 1):
            for col in range(min_col, max_col + 1):
                if image[row][col] == current_color:
                    run_length += 1
                    if current_color == val:  # Increment pixel
                        pixel_count += 1
                else:
                    rle_encoding.append(run_length)
                    current_color = image[row][col]
                    run_length = 1

        rle_encoding.append(run_length)
        self.start_value = start_value
        self.rle_encoding = rle_encoding
        self.pixel_count = pixel_count  # Store pixel count
        return start_value, rle_encoding, pixel_count
```

Output: (space saved)

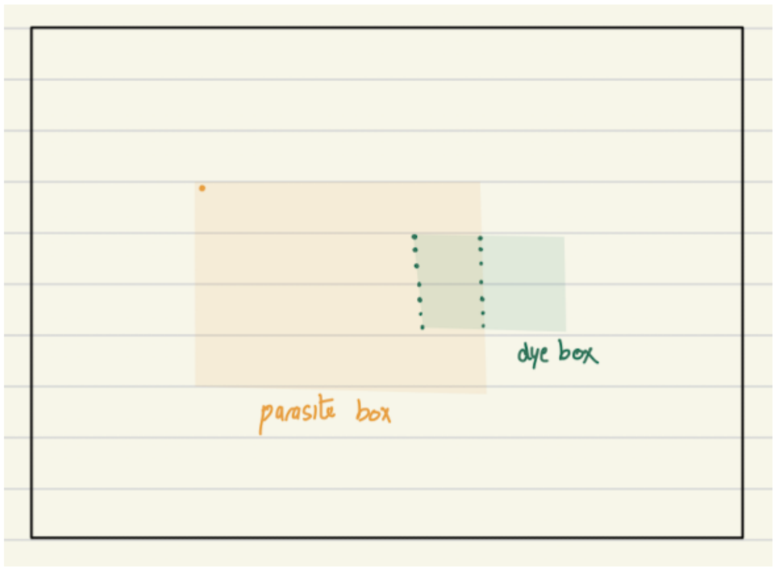
```
size of original image:  100000128 bits
size of compressed image:  108005  bits
```

Detecting Cancer

- Step (1): Computing overlapped bounding box: We now only have to worry about the pixels in this box rather than checking all the pixels in the image

```
def compute_overlapped_box(self):
        if self.microscope_image is None or self.dye_image is No
            raise ValueError("Both microscope and dye images mus

        # Extract bounding boxes
        microscope_bbox = self.microscope_image[0]
        dye_bbox = self.dye_image[0]

        # Calculate intersection of bounding boxes
        min_row = max(microscope_bbox[0][0], dye_bbox[0][0])
        min_col = max(microscope_bbox[0][1], dye_bbox[0][1])
        max_row = min(microscope_bbox[1][0], dye_bbox[1][0])
        max_col = min(microscope_bbox[1][1], dye_bbox[1][1])

        #print(((min_row, min_col), (max_row, max_col)))
        # Check if intersection is valid
        if min_row <= max_row and min_col <= max_col:
            overlapped_box = ((min_row, min_col), (max_row, max_
            self.overlapped_box = overlapped_box
            self.curr_rl_microscope = self.compute_run_distance_
            self.curr_rl_dye = self.compute_run_distance_to_inte

            return overlapped_box
        else:
            return None
```

- Step(2): Compute the number of pixels that overlap between a parasite and the dye
    - Compute the run distance to the overlap for both images.
    - Iterate through the run lengths and toggle the value, to get the corresponding pixel value. If both pixels are non-zero, then dye is on the parasite. Count it.
    - Everytime we reach a new row of pixels, the run distance to that pixels has to be recomputed.



- Step (3): detect cancer. If the number of overlapping pixels is more than 10% of the number of parasite pixels, then the parasite has cancer.

Output:

```
Percentage of dye coverage on parasite: 0.152
Cancer detected
Percentage of dye coverage on parasite: 0.147
Cancer detected
Percentage of dye coverage on parasite: 0.030
No cancer
Percentage of dye coverage on parasite: 0.061
No cancer
Percentage of dye coverage on parasite: 0.010
No cancer
```