Nikki Liu June 2024

Dragonfruit AI Challenge

1. Efficient Data Structures for Image Representation

For storing and processing the images, we need efficient data structures that can handle the high resolution (100,000 x 100,000 pixels) while minimizing storage space.

Microscope Images

We are given that microscope images typically have large contiguous regions of black pixels (background) and a single contiguous blob of white pixels (parasite).

Using a **Run-Length Encoding** (RLE) is an efficient way to store images like these.

RLE then compresses the image by storing the lengths of runs of consecutive pixels with the same value.

RLE Representation for Microscope Image:

- Store the pixel value (black (0) or white (1)) and the length of the run.
- Example: [(0, 500), (1, 2500), (0, 97500)] for a row with 500 black pixels, 2500 white pixels, and 97500 black pixels.

Estimated Storage Size:

- In the worst case (an image with alternating black and white pixels), the storage size would be around **100,000 * 2** entries (each run represented by a pair of values), resulting in 200,000 values.
- If each value takes 4 bytes (assuming integer representation), the total size would be 800,000 bytes per row, leading to approximately **80 GB** for the entire image.

We also know that Dye Sensor Images can have more scattered lit areas. Therefore, using a **Sparse Matrix Representation** can be more efficient. We will store only the coordinates of the lit pixels (where dye is present).

Sparse Matrix Representation for Dye Sensor Image:

- Store the coordinates of lit pixels.
- Example: [(x1, y1), (x2, y2), ...] where each tuple (x,y) represents the position of a lit pixel.

Estimated Storage Size:

- Assuming fewer than 10% of pixels are lit, this results in at the very most **10,000,000** coordinates.
- If each coordinate pair takes 8 bytes (4 bytes each for x and y), the total size would be 80,000,000 bytes or **80 MB** for the entire image.

2. Generating Simulated Images

We can simulate images using the above data structures in python:

generate.py:

```
import random

def generate_microscope_image(width, height):
    image = []
    for _ in range(height):
    row = []
    for _ in range(width):
    if random.random() < 0.75:  # 75% chance for background (black pixel)
    row.append(0)
    else:  # 25% chance for blob (white pixel)
    row.append(1)
    image.append(row)
    return image

def generate_dye_image(width, height):
    image = []
    for _ in range(height):
    row = []
    for _ in range(width):
    if random.random() < 0.05:  # 5% chance for lit pixel (dye present)
    row.append(1)
    else:
    row.append(0)
    image.append(row)
    return image

if __name__ == "__main__":
    width, height = 100, 100  # Using smaller dimensions for testing
    microscope_image = generate_microscope_image(width, height)
    dye_image = generate_dye_image(width, height)

    # Save generated images to files
    with open("microscope_image.txt", "w") as f:
```

```
for row in microscope_image:
f.write(','.join(map(str, row)) + "\n")

with open("dye_image.txt", "w") as f:
for row in dye_image:
f.write(','.join(map(str, row)) + "\n")

print("Images generated and saved.")
```

Explanation:

generate_microscope_image:

- This creates an image with a black background and a white blob, represented as a 2D list. Each pixel in the list is either 0 (black, representing the background) or 1 (white, representing the blob).

generate_dye_image:

- This creates an image with randomly scattered lit pixels, represented as a 2D list. Each pixel is either 1 (dye present) or 0 (no dye).

Main:

- Uses the generate_microscope_image and generate_dye_image functions to generate two images.
- Saves generated images to two text files (microscope_image.txt and dye_image.txt). Each row of the image is now saved as a comma-separated string of pixel values.

3. Detecting Cancer in Parasites

This function calculates whether the amount of dye within the parasite exceeds 10% of its total area.

detect_cancer.py:

```
def has_cancer(microscope_image, dye_image, width, height):
    total_blob_pixels = 0
    total_dye_pixels = 0

    for y in range(height):
    for x in range(width):
    if microscope_image[y][x] == 1:  # Blob pixel
    total_blob_pixels += 1
    if dye_image[y][x] == 1:  # Dye within blob
        total_dye_pixels += 1

    dye_percentage = total_dye_pixels / total_blob_pixels if total_blob_pixels > 0 else 0
    return dye_percentage > 0.10

if __name__ == "__main__":
    # Load generated images from files
    with open("microscope_image.txt", "r") as f:
    microscope_image = [list(map(int, line.strip().split(','))) for line in f]

    with open("dye_image.txt", "r") as f:
    dye_image = [list(map(int, line.strip().split(','))) for line in f]

    width, height = len(microscope_image[0]), len(microscope_image)  # Get dimensions from image
    result = has_cancer(microscope_image, dye_image, width, height)
    print(f"Parasite has cancer: {result}")
```

Explanation:

- The has_cancer function reads the images from the files, splitting each line by commas to reconstruct the 2D lists.
- The dimensions of the images are determined by the length of the lists.
- The function checks for total_blob_pixels > 0 to avoid division by zero.
- Main loads the generated picture from the file

4. Optimizing Execution Speed

We optimize the execution speed using NumPy

optimize.py:

```
import numpy as np

def has_cancer_optimized(microscope_image, dye_image, width, height):
        # Convert lists to NumPy arrays for efficient processing
        microscope_array = np.array(microscope_image)
        dye_array = np.array(dye_image)

        # Count the number of blob pixels and dye pixels within the blob
        blob_pixels = np.sum(microscope_array == 1)
        dye_pixels_within_blob = np.sum((microscope_array == 1) & (dye_array == 1))

        # Calculate the percentage of dye pixels within the blob
        dye_percentage = dye_pixels_within_blob / blob_pixels if blob_pixels > 0 else 0
        return dye_percentage > 0.10

if __name__ == "__main__":
        # Load generated images from files
        with open("microscope_image.txt", "r") as f:
        microscope_image = [list(map(int, line.strip().split(','))) for line in f]

        with open("dye_image.txt", "r") as f:
        dye_image = [list(map(int, line.strip().split(','))) for line in f]

        width, height = len(microscope_image[0]), len(microscope_image)  # Get dimensions from image
        result = has_cancer_optimized(microscope_image, dye_image, width, height)
        print(f"Parasite has cancer: {result}")
```

Explanation: We pretty much do the same thing as detect_cancer but with NumPy arrays

- Converts the 2D lists of the microscope and dye images to NumPy arrays.
- Counts the number of blob pixels (parasite body) in the microscope image.
- Counts the number of dye pixels within the blob area using a logical AND operation between the two arrays.
- Calculates the percentage of dye pixels within the blob.
- Returns True if the dye percentage exceeds 10%, indicating cancer; otherwise, returns False.

5. Other Compression Techniques and Their Impact

Compression Techniques:

1. **Block-Based Compression**:
   - **Method**: Divide the image into smaller blocks and compress each block individually.
   - **Impact**: Reduces storage space by exploiting redundancy within each block but may introduce slight decompression overhead.
2. **Wavelet Transform**:
   - **Method**: Apply wavelet transformation to compress the image by removing less significant details.
   - **Impact**: Significantly reduces storage space while maintaining important features. Although it introduces more computational complexity during compression and decompression.
3. **PNG Compression**:
   - **Method**: Use the PNG format, which combines lossless compression algorithms
   - **Impact**: Efficiently reduces storage size with minimal impact on runtime.

Storage and Runtime Costs:

- **Typical Storage Costs**:
  - **Uncompressed Image**: Each 100,000 x 100,000 pixel image requires approximately 10 GB of storage (assuming 1 byte per pixel).
  - **Compressed Image**: Compression techniques can reduce storage requirements by an order of magnitude. For example, block-based compression or PNG might reduce the size to 1-2 GB.
- **Runtime Costs**:
  - Compression and decompression add computational overhead, which can be minimized with efficient algorithms.
  - The actual runtime for processing a 100,000 x 100,000 pixel image using optimized techniques (like the NumPy-based approach) is manageable

6. Tools and Techniques Used to Solve the Challenge

Tools Used:

- **Python**: The primary programming language used for my implementation.
- **NumPy**: A library for numerical computations, used for efficient array operations and processing large images.

LLM Techniques:

- **Large Language Models (LLMs)**: Used for initial drafting and explanation of concepts, ensuring clarity and correctness in the description of algorithms and processes.

Steps to Solve the Challenge:

1. **Understanding Requirements**:
    - Analyze the requirements to determine the need for efficient storage and processing of high-resolution images.
2. **Choosing Data Structures**:
    - Select appropriate data structures (e.g., 2D lists, NumPy arrays) for representing images to balance storage efficiency and processing speed.
3. **Generating Simulated Images**:
    - Implement functions to generate realistic simulated images for testing purposes.
4. **Cancer Detection Algorithm**:
    - Develop and optimize the cancer detection algorithm using NumPy for efficient processing.
5. **Testing and Verification**:
    - Test the implementation with simulated images to ensure accuracy and performance.