Image Segmentation Report

Our team has completed an initial analysis using the max flow/min-cut theorem for image segmentation. We utilized linear programming (LP) techniques to optimize the segmentation process. In the following sections, we will discuss methodology, provide code snippets demonstrating the application of our approach, and share example images with optimal cuts.

Problem Statement:

Our objective was to distinguish the background from the foreground in grayscale images by modeling the image as a flow network. In this network, each node represents an element, and edges between nodes carry flow with a capacity that limits how much flow can pass. We then severed the weakest connections with the Max Flow / Min Cut theorem, creating a clear divide between foreground and background.

Overview of the Max Flow/Min Cut Theorem:

The Max Flow / Min Cut Theorem is highly relevant to image segmentation problems. We used this theorem because it can efficiently separate regions of an image based on pixel intensity differences.

For this project, individual pixels are treated as nodes and the edges symbolize the connections between adjacent pixels. The capacities of these edges indicate the probability that neighboring pixels belong to the same segment. Regarding the foreground and background, the source node (starting point of the flow) denotes the foreground pixels, while the sink node (endpoint of the flow) denotes the background. After developing the network described below, we identified the minimum cut that effectively separates the foreground pixels from the background pixels.

Creating the network:

We first converted the image into a grayscale image and then into a flow network. In this network, each pixel is treated as a node in a grid, and edges are created between neighboring pixels based on the similarities of their intensity values. These connections are weighted according to how similar the pixel intensities are.

Here is our formula to calculate the similarity between neighboring pixels:

$$\text{similarity} = 100 \times \exp\left(\frac{-(I_i - I_j)^2}{2\sigma^2}\right)$$

I_i and I_j represent the intensities of pixels i and j, and σ controls how much we weigh the differences between pixel intensities. In our analysis, we automate the sigma to adjust it based on the image uploaded.

In addition to the pixel nodes, we introduced the source (foreground pixels) and sink (background pixels) nodes, further discussed in later sections.

Framing a Linear Programming Problem:

We initially started by working on simpler images to develop the code's framework, training our model on the 20x20 box and oval images shown below. These “images” were actually contained in .csv files with cells arranged in a 2D grid of pixel intensities. We have labeled each step in the code, and will now explain what each step accomplishes.

Step 1: Loading and preprocessing the File

In this step, the *CSV* images are inputted and then converted into a pandas data frame, essentially becoming a matrix of pixel intensities. The intensities are then normalized from values of 0-255 to values between 0 and 1. Normalization ensures consistency across different images by improving numerical stability during further calculations.

Code:

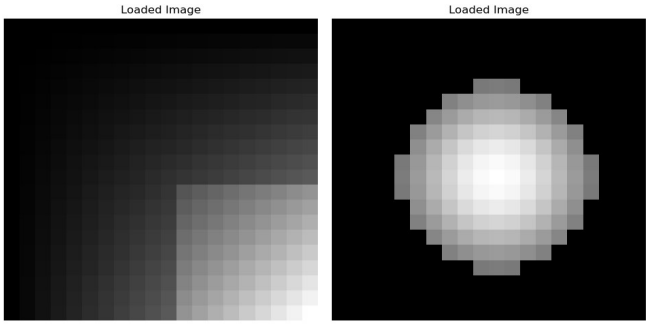
```
def load_file(file_path):
    file_extension = os.path.splitext(file_path)[1].lower()

    if file_extension == '.jpg':
        img = Image.open(file_path).convert('L')  # Convert JPEG to grayscale
        return np.array(img) / 255.0  # Normalize to [0, 1]

    elif file_extension == '.csv':
        df = pd.read_csv(file_path, header=None)  # Load CSV as DataFrame
        return df.values / 255.0  # Normalize to [0, 1]

    else:
        raise ValueError("Unsupported file format. Use .jpg or .csv.")
```

Result:



Step 2: Preprocessing the Image

After the image is successfully loaded, the dimensions of the image (number of rows and columns) are extracted. This allows us to loop through specific pixels in the image grid when creating the flow network and identify neighboring pixels.

Code:

```
if file_path.endswith('.jpg'):
    img = Image.fromarray((image * 255).astype(np.uint8))  # Convert back to Image for enhancement
    enhancer = ImageEnhance.Contrast(img)
    img_enhanced = enhancer.enhance(1.5)  # Adjust enhancement factor
    image = np.array(img_enhanced) / 255.0  # Normalize after enhancement

    # Display after contrast enhancement
    plt.figure(figsize=(6, 6))
    plt.imshow(image, cmap='gray', origin='upper')
    plt.title('After Contrast Enhancement')
    plt.axis('off')
    plt.show()

# Step 3: Preprocess the image/CSV
num_rows, num_cols = image.shape
```

Step 3: Specifying Background and Foreground Pixels

After creating the network, we specify the background and foreground pixels. To start, all pixels on the borders (top, bottom, left, and right edges) of the image are considered background pixels. Then, we use Otsu's thresholding method to find the optimal threshold for separating the foreground from the background based on pixel intensity. The images show that pixels with intensity above the threshold are deemed foreground pixels (green dots).

Code:

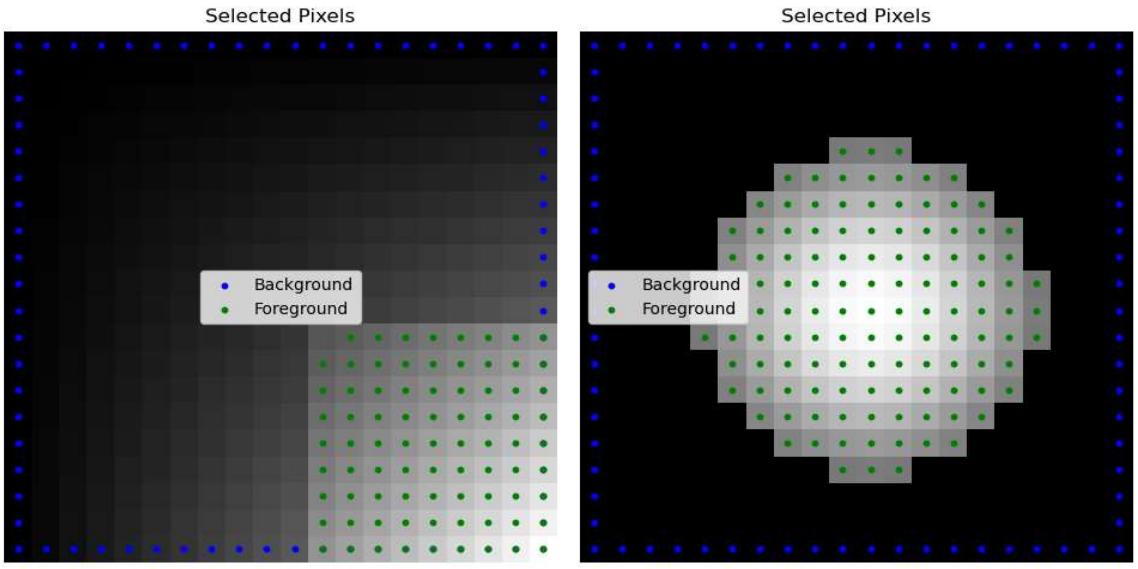
```
# Background pixels: All border pixels
border_pixels = []
for row in range(num_rows):
    for col in range(num_cols):
        if row == 0 or row == num_rows - 1 or col == 0 or col == num_cols - 1:
            border_pixels.append((row, col))

# Compute Otsu's threshold for foreground selection
threshold = threshold_otsu(image)
print(f"Computed Otsu threshold: {threshold}")

# Foreground pixels: Pixels with intensity above the threshold
foreground_pixels = [
    (row, col) for row in range(num_rows)
    for col in range(num_cols)
    if image[row, col] > threshold
]

# Plot the selected background and foreground pixels
plt.figure(figsize=(6, 6))
plt.imshow(image, cmap='gray', origin='upper')
plt.scatter(
    [col for row, col in border_pixels],
    [row for row, col in border_pixels],
    color='blue',
    s=10,
    label='Background'
)
```

Result:



Step 4: Constructing the Flow Network

The image is treated as a grid of connected pixels, each representing a node in the network. The total number of nodes equals the number of pixels, plus two special nodes acting as the source and sink. The source defines the starting point (the foreground pixel connected to all other foreground pixels). In contrast, the sink represents the endpoint (the background pixel connected to all other background pixels) for the flow in the network. By introducing these source and sink nodes, we established the flow direction through the network and defined the boundaries between regions.

Subsequently, we calculated the intensity differences between neighboring pixels to assess the strength of connections (similarity) between pixels. Additionally, in choosing our sigma, we used the standard deviation of these intensity differences, allowing for a more dynamic selection of the sigma based on the image. Using the similarity formula provided earlier, we calculated and stored the similarity between each pixel and its neighbors in a network matrix. Higher similarity between two pixels indicates a greater likelihood of the pixel belonging to the same region.

Code:

```
# Compute sigma as the standard deviation of intensity differences
sigma = np.std(intensity_diffs)
print(f"Computed sigma: {sigma}")

# Initialize the network capacity matrix
network = np.zeros((total_nodes, total_nodes), dtype=float)

# Compute similarities and fill the network matrix
max_similarity = 0
for row in range(num_rows):
    for col in range(num_cols):
        idx = pixel_index(row, col)
        Ii = image[row, col]
        neighbors = []
        if row > 0:
            neighbors.append((row - 1, col))
        if row < num_rows - 1:
            neighbors.append((row + 1, col))
        if col > 0:
            neighbors.append((row, col - 1))
        if col < num_cols - 1:
            neighbors.append((row, col + 1))
        for n_row, n_col in neighbors:
            n_idx = pixel_index(n_row, n_col)
            Ij = image[n_row, n_col]
            intensity_diff = Ii - Ij
            similarity = np.ceil(100 * np.exp(- (intensity_diff) ** 2 / (2 * sigma ** 2)))
            similarity = float(similarity)
            network[idx, n_idx] = similarity
            network[n_idx, idx] = similarity  # Undirected network
            if similarity > max_similarity:
                max_similarity = similarity

# **Swap the source and sink connections**
# Connect all foreground pixels to the source
for row, col in foreground_pixels:
    idx = pixel_index(row, col)
    network[source, idx] = max_similarity

# Connect all border pixels to the sink
for row, col in border_pixels:
    idx = pixel_index(row, col)
    network[idx, sink] = max_similarity
```

Step 5: Maximizing the Flow with Gurobi

Using Gurobi, we created a linear programming model intending to maximize the flow from the source node to the background pixels. Flow variables are created for each edge (connection between two pixels) in the network. Flow conservation constraints are added to ensure that the flow entering a pixel equals the flow leaving the pixel, except for the source and sink. The model is then solved to find the maximum flow from the source to the sink.

Code:

```
model = gp.Model("Image_Segmentation")
model.Params.OutputFlag = 0  # Suppress solver output
# Identify all edges with positive capacity
edges = np.argwhere(network > 0)
num_edges = edges.shape[0]
# Add flow variables for each edge
flows = model.addVars(num_edges, lb=0, ub=network[edges[:,0], edges[:,1]], name='flow')
# Set objective: maximize flow from source to sink
source_edges = [i for i, (u, v) in enumerate(edges) if u == source]
model.setObjective(gp.quicksum(flows[i] for i in source_edges), GRB.MAXIMIZE)
# Add flow conservation constraints for each node except source and sink
for node in range(total_nodes):
    if node == source or node == sink:
        continue
    in_edges = [i for i, (u, v) in enumerate(edges) if v == node]
    out_edges = [i for i, (u, v) in enumerate(edges) if u == node]
    model.addConstr(
        gp.quicksum(flows[i] for i in in_edges) == gp.quicksum(flows[i] for i in out_edges),
        name=f'flow_conservation_node_{node}'
    )
# Optimize the model
model.optimize()
```

Step 6: Identifying and Visualizing Cuts

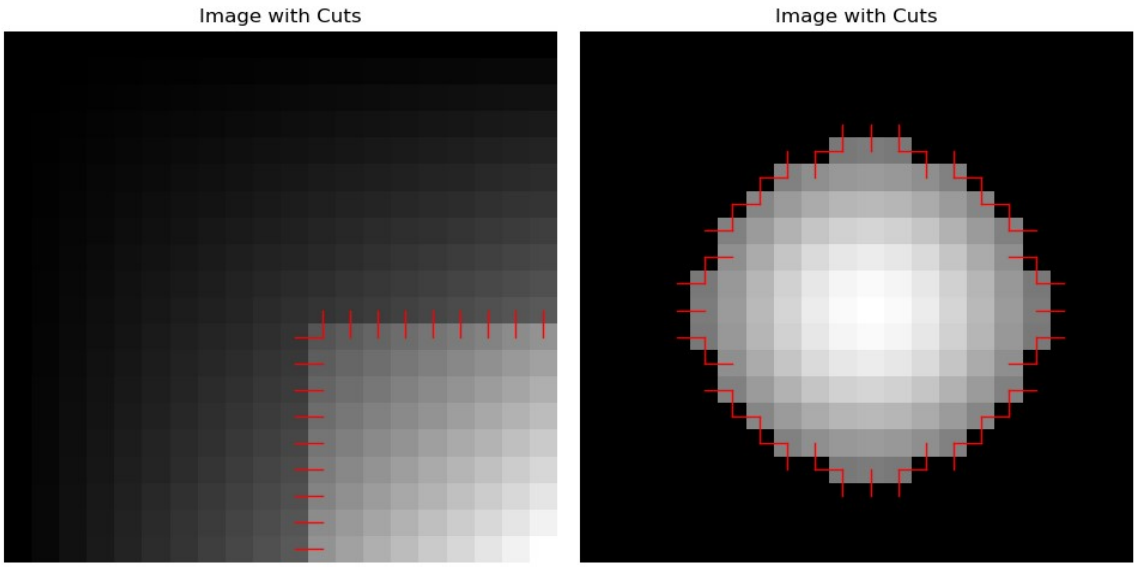
After solving the LP model, the residual network is constructed by subtracting the actual flow from the maximum possible flow on each edge. Subsequently, a depth-first search (DFS) is performed on the residual network starting from the source to find all nodes that remain reachable from the source. The edges connecting reachable nodes to non-reachable nodes

become the cuts. These cuts are represented as red lines on the plot, indicating where the image has been segmented.

Code:

```
if model.status == GRB.OPTIMAL:
    print(f"Maximum flow value: {model.objVal}")
    # Extract flow values
    flow_values = model.getAttr('X', flows)
    # Build residual network
    residual_network = np.copy(network)
    for i in range(num_edges):
        u, v = edges[i]
        flow = flow_values[i]
        residual_network[u, v] -= flow
    # Perform DFS to find reachable nodes from source in residual network
    visited = [False] * total_nodes
    def dfs(u):
        visited[u] = True
        neighbors = np.where(residual_network[u, :] > 1e-6)[0]
        for v in neighbors:
            if not visited[v]:
                dfs(v)
    dfs(source)
    # Identify cut edges: edges from visited to unvisited nodes
    cuts = []
    for i in range(num_edges):
        u, v = edges[i]
        if visited[u] and not visited[v]:
            cuts.append((u, v))
    print(f"Number of cuts found: {len(cuts)}")
    # Sum the capacities of the cuts
    cut_capacity = sum(network[u, v] for u, v in cuts)
    print(f"Total cut capacity: {cut_capacity}")
    if abs(model.objVal - cut_capacity) < 1e-6:
        print("Max flow equals min cut capacity (verified).")
```

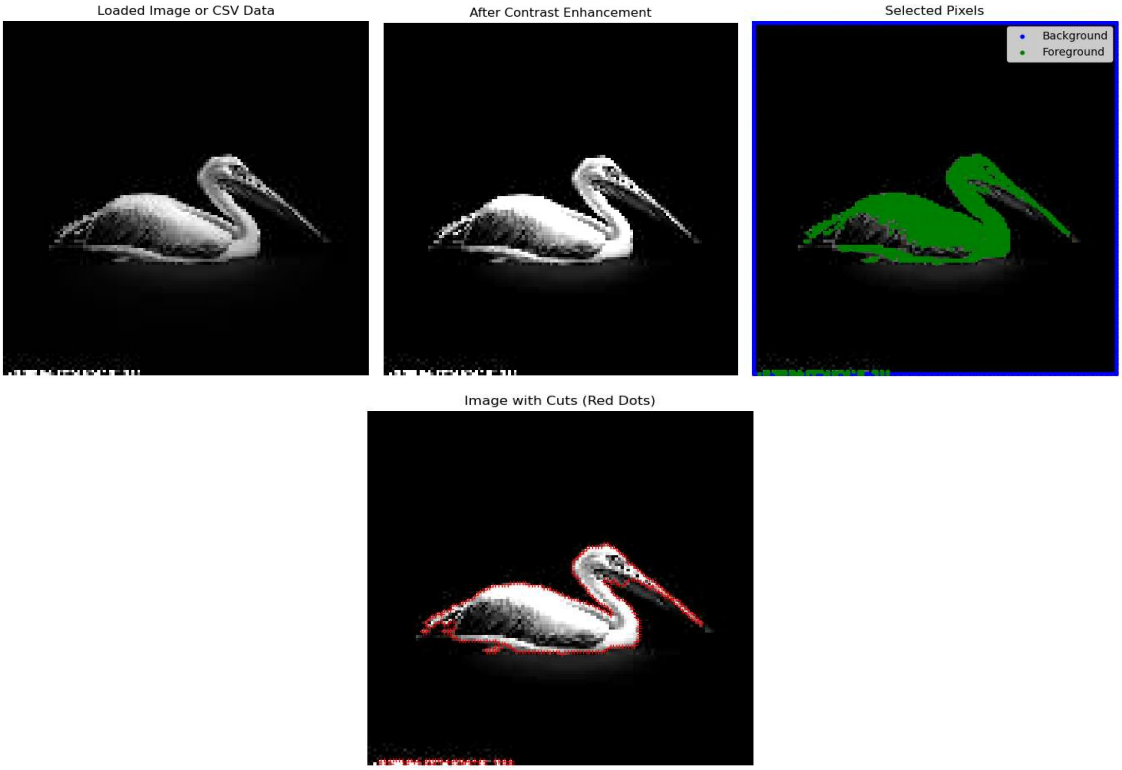
Result:



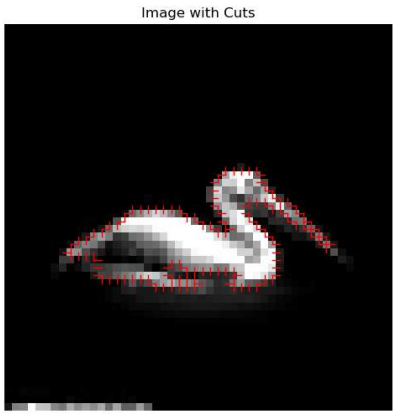
Segmenting JPG images

After successfully getting the code to work for simple images, we tested it on a more complex, 128x128 *jpg* image of a pelican to ensure its compatibility with any image. Naturally, since our code was initially constructed with processing only *CSV* files in mind, we also had to make adjustments to process *JPG* files.

Our first change was ensuring our code could process *CSV* and *JPG* files. By checking the file extension when loaded, we process the different types in slightly different ways. We load *CSV* files as *Pandas* data frames, while we use the PIL library to load JPGs instead. We ensured the image was in grayscale for JPG images by converting it immediately after processing. Additionally, for JPG images only, we increased the contrast of the image by 1.5x to better define the foreground and background before cutting.



Optionally, we also tested a resized 50x50 image of the pelican, shown below. This is because the segmentation on the full-resolution pelican image can take 15+ minutes, whereas the resized image will only take about a minute. We also included the resized pelican code for further cohesion in our process.



Challenges and Considerations

We encountered two major issues during our project. Firstly, we experienced long loading times for images, such as the 128x128 JPG pelican, which took over 10 minutes to load. As the image size increased, the number of nodes and edges in the flow network also increased, leading to longer processing times. To address this, if given more time, we would aim to create a more efficient data structure using advanced graph libraries, like NetworkX, which offer optimized implementations of max flow/min cut algorithms to reduce processing time.

Secondly, we encountered challenges when dealing with complex image features. Our initial code performed well with simpler images like the box and oval, but we faced difficulties when working with more complex images containing texture and shading, such as the pelican. To improve this issue, we enhanced the pixel similarity metric.

Conclusion

We successfully implemented the Max Flow/Min Cut theorem for image segmentation, applying LP concepts to distinguish the foreground from the background. By normalizing pixel intensities, and computing similarities between neighboring pixels, we built a flow network. Then, using Gurobi, we solved the maximum flow problem and identified optimal cuts to separate foreground from background for both simple 20x20 and 128x128 images. Although we encountered issues regarding processing time, this is a great V1 to iterate off of. Future enhancements will aim to improve performance and enhance pixel similarity metric for both black/white and colored images.

Group 6 – Karthik Thiagarajan, Mason Shu, Siboney Cardoso, Agnitra Das