IMAGE SEGMENTATION

By: Kimble Horsak, Samuel Chen, Andrew White, and Jose Currea

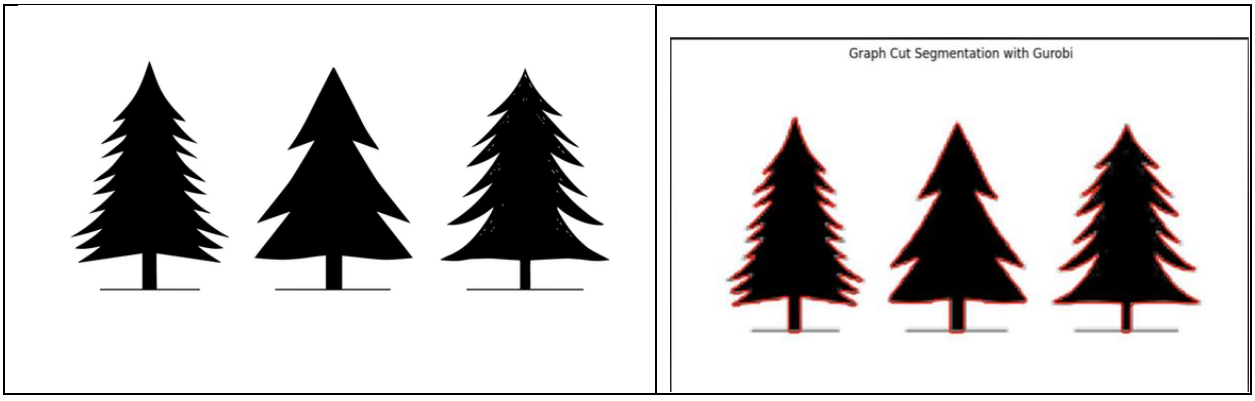
This report explores the use of the Max Flow / Min Cut Theorem in image segmentation. We will apply this method to segment images by finding optimal cuts that separate different regions based on intensity values. This method is implemented using Linear Programming (LP) techniques.

Before beginning the analysis, it is important to recall what this theorem is. This theorem states that “the maximum amount of flow passing from the source to the sink is equivalent to the net flow of the edges in the minimum cut. So by solving the max-flow problem, we directly solve the min-cut problem as well”.

Let’s illustrate this theorem. Given the fact that our startup is developing a new feature to help users quickly separate different objects in an image, like separating a person from the background. Here's how the Max Flow / Min Cut Theorem applies:

- Max flow problem:
    - Wanting to separate a person from the background in a photo. In this case, the "source" is all the pixels that the user believes belong to the person, and the "sink" is all the pixels the user believes that belong to the background.
    - Our task is to connect as many pixels from the person to the source and as many background pixels to the sink. We must be careful when dealing with adjacent pixels given the fact that we must respect each pixel’s intensity.
    - We need to maximize the number of pixels from the person (source) and the number of pixels for the background (sink).
- Min cut problem
    - Cutting the least number of pixel connections so that the person is completely separated from the background.
    - We need to minimize the number of connections (pixel boundaries) that will separate the person from the background effectively. This "cut" is the boundary around the person, separating them from the background.
    - The minimum cut gives us the optimal boundary where the person's pixels meet the background's pixels.

THESE SCENARIOS WILL LOOK SOMETHING LIKE THIS:



Notice how we have three trees and how we are separating them from the background.

For our implementation of this theorem, we used several steps to deal with this theorem. The steps were:

1. Computing Sigma (Calculating similarity among pixels)
2. Finding Boundary Pixels
3. Adding source (main image) and sink (background) nodes
4. Setting up the optimization problem
5. Create objective functions and constraints

Step 1. Calculate similarity

```
def compute_sigma(box):
    intensity_diffs = []
    n_rows, n_cols = box.shape
    for r in range(n_rows):
        for c in range(n_cols):
            neighbors = [
                (r-1, c-1), (r-1, c), (r-1, c+1),
                (r, c-1),           (r, c+1),
                (r+1, c-1), (r+1, c), (r+1, c+1)
            ]
            for nr, nc in neighbors:
                if 0 <= nr < n_rows and 0 <= nc < n_cols:
                    intensity_diffs.append(abs(box[r, c] - box[nr, nc]))
    sigma = np.std(intensity_diffs)
    print(intensity_diffs)
    return sigma if sigma != 0 else 0.1
```

This function checks how similar the brightness of pixels is in a small part of an image. It looks at each pixel and compares its brightness to the pixels around it. If most of the pixels have similar brightness, the function will return a small value, meaning the area is smooth. If there's a lot of difference in brightness between the pixels, the function will return a larger value, indicating the area has more texture or is less smooth. It's like checking how "bumpy" or "flat" the brightness is in that part of the image.

Step 2. Find Boundary Pixels

```
# Function to detect boundary pixels using adaptive thresholding and edge detection
def find_boundary_pixels(box):
    # Adaptive thresholding
    thresh = threshold_otsu(box)
```

```
    binary_box = box > thresh

    # Edge detection
    edges = feature.canny(box, sigma=2)
    n_rows, n_cols = box.shape
    boundary_pixels = []
    for r in range(n_rows):
        for c in range(n_cols):
            if edges[r, c]:
                boundary_pixels.append(r * n_cols + c)
    return boundary_pixels

# Get boundary pixels
boundary_pixels = find_boundary_pixels(box)

# Ensure we have boundary pixels
if not boundary_pixels:
    raise ValueError("No boundary pixels found. Please check the image content.")

# Define the source pixel (first boundary pixel)
source_pixel =  boundary_pixels[0]   #Can replace with a number within the sixe of the box array, ei 128x128 can
replace with 1 - 16384
print("boundary pixel")
print(boundary_pixels[0])
```

This function is like a "detective" for finding the edges or outlines in a small part of an image, called box. First, it uses a method called adaptive thresholding to decide which pixels are brighter or darker compared to the average brightness in the image. Then, it uses another method called edge detection to find the pixels where there's a big change in brightness, like the border between an object and its background.

After finding these edge pixels, it goes through the whole image area to collect their positions in a list called boundary_pixels. If there are no such edge pixels found, it will give an error saying, "No boundary pixels found. Please check the image content," meaning the image might be too plain or uniform. Finally, it selects the first pixel in the list as the starting or "source" pixel, which you can also manually choose by giving a number that fits within the image size, like picking a number between 1 and 16384 for an image that is 128x128 in size. This is useful for focusing on the specific boundary area in an image.

We define the source node based on our edge detection function and simply use the first index of our boundary pixels with source_pixel = boundary_pixels[0] use, however this source pixel value can be hard coded to an

individual pixel as long is it is within the range of the box matrix. For example for a 128 x 128 box matrix the source pixel can be defined as any value between 1 and 16,384 (128*128). Sigma is computed based on the standard deviation of neighboring brightness difference, which helps tune the model to detect similarities based on different levels of contrast. Since sigma is inversely proportional to the threshold to determine a cut, a lower sigma yields more cuts, which is helpful for images with a wide range of brightnesses.

Step 3. Perform Graph Cuts

```
# Function to perform graph cut segmentation using Gurobi
def graph_cut_segmentation(box):
```

This function segments an image into foreground and background regions using graph cut optimization. It starts by calculating the variation in pixel intensities (sigma) and then sets up a graph model with Gurobi, where each pixel is a node connected to either a source (representing the foreground) or a sink (representing the background). It creates edges between pixels and assigns weights based on intensity differences. The function then solves an optimization problem to find the maximum flow from the source to the sink, which corresponds to the optimal cut between foreground and background. Finally, it reconstructs the segmented image based on the computed flow and displays the result.

Explanation of the cuts

We find the cuts by finding where the network has been completely saturated. This is found by getting the residual capacities, which is defined as the capacity of the edge minus the flow through the edge. These edges are then found through iterating through the entire network and finding the edges with no residual capacity and making cuts there.
This works in the context of maximum flow because the fully saturated edges are where the flow bottlenecks are. In order to decrease the saturation of those edges to give them residual capacity, you would need to decrease flow elsewhere in the network, which effectively produces a non optimal solution to the objective function. Thus, the set of edges that have full saturation
matches the locations where cuts need to be made

```
# Now create flow variables for all edges in capacities
    for (i, j), cap in capacities.items():
        flow_vars[(i, j)] = model.addVar(lb=0, ub=cap, vtype=GRB.CONTINUOUS, name=f'f_{i}_{j}')

    model.update()

    # Flow conservation constraints for all nodes except source and sink
    for idx in range(n_pixels):
        inflow = gp.quicksum(flow_vars[(i, idx)] for i in range(total_nodes) if (i, idx) in flow_vars)
        outflow = gp.quicksum(flow_vars[(idx, j)] for j in range(total_nodes) if (idx, j) in flow_vars)
```

```
    model.addConstr(inflow == outflow, name=f'flow_conservation_{idx}')

# Objective: Maximize total flow from source to sink
total_flow = gp.quicksum(flow_vars[(source, j)] for j in range(total_nodes) if (source, j) in flow_vars)
model.setObjective(total_flow, GRB.MAXIMIZE)

# Solve the model
model.optimize()

# Reconstruct the segmentation
if model.status == GRB.OPTIMAL:
    # Build residual graph and perform BFS from source
    residual_capacities = {}

    for (i, j), var in flow_vars.items():
        flow = var.X
        cap = var.UB
        residual_capacity = cap - flow
        if residual_capacity > 1e-5:  # Tolerance to avoid floating point errors
            residual_capacities.setdefault(i, []).append(j)
        if flow > 1e-5:
            residual_capacities.setdefault(j, []).append(i)  # Reverse flow
```

The Gurobi implementation in the function sets up a graph-based optimization model where each pixel in the image is treated as a node, connected to either a source node (representing the foreground) or a sink node (representing the background), with edges between them representing possible pixel connections. Flow variables are created for each edge, bounded by capacities that depend on the intensity differences between connected pixels. The objective is to maximize the total flow from the source to the sink, which effectively determines the optimal cut separating foreground and background pixels. Flow conservation constraints ensure that the flow through each pixel node is balanced, and Gurobi's optimizer is used to find this maximum flow, which corresponds to the desired segmentation of the image.

In conclusion, this implementation uses the max-flow/min-cut theorem and Gurobi's optimization to effectively segment images. Each pixel is treated as a point in a network, and the function sets up connections between them based on their brightness differences. By finding the best way to "cut" the network—essentially the easiest way to separate the source and sink nodes—we can identify the most natural division between the foreground and background. This method ensures that the maximum flow through the network equals the minimum effort to separate it, giving us a clear boundary for segmentation. It's a great example of how complex math and optimization can solve real-world problems like image processing.

SOME IMPLEMENTATIONS WE TRIED: