Saaket Joshi (ssj778), Hadley Krummel (hek454), Mikala Lowrance (mll3774), Twinkle Panda (

Professor Mitchell

Optimization

29 September 2024

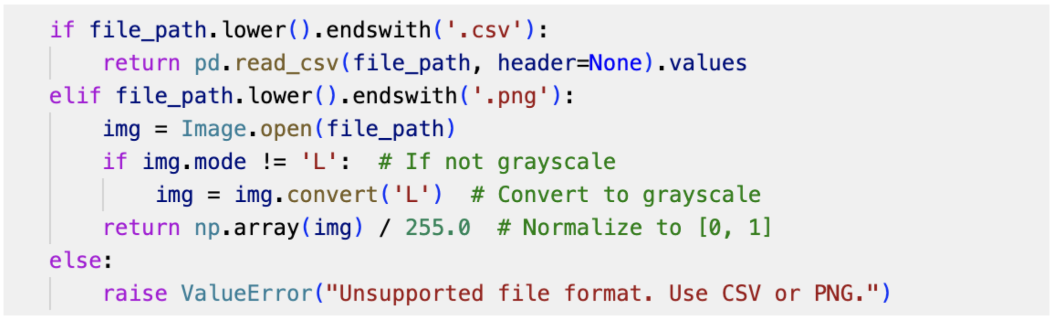
Optimization Image Segmentation Report

Problem Statement

The problem is focused on image segmentation using a max flow/minimum cut algorithm. The goal is to separate the foreground from the background in an image by constructing a graph where each pixel is a node, and edges represent the similarity between neighboring pixels. By solving the max flow problem, the algorithm finds the minimum cut, which represents the optimal boundary between the foreground and background, effectively segmenting the image. This method combines graph theory and optimization to achieve accurate image partitioning for tasks like object recognition and scene analysis.

Image Preprocessing

The first step required was to ensure that the program could read a csv or png image. Additionally, if the image is not already in grayscale, then the program converts the image into a grayscale image and normalizes the pixels to values between 0 and 1 in a numpy array.



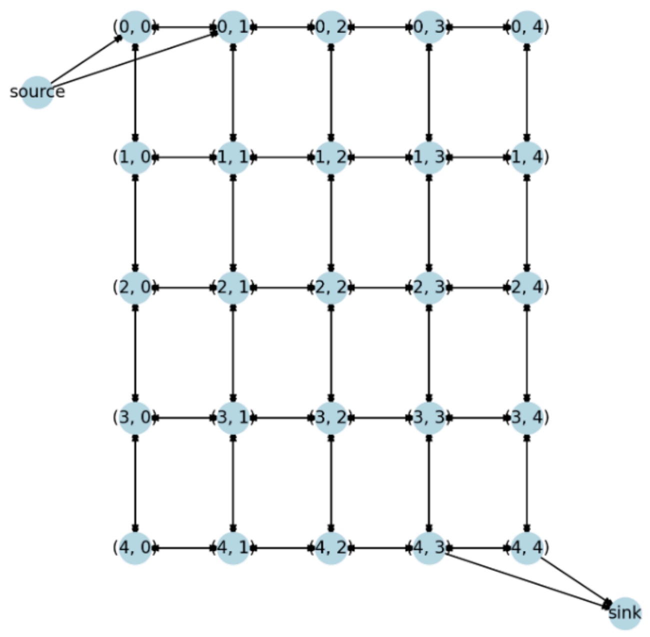
```python
if file_path.lower().endswith('.csv'):
    return pd.read_csv(file_path, header=None).values
elif file_path.lower().endswith('.png'):
    img = Image.open(file_path)
    if img.mode != 'L':  # If not grayscale
        img = img.convert('L')  # Convert to grayscale
    return np.array(img) / 255.0  # Normalize to [0, 1]
else:
    raise ValueError("Unsupported file format. Use CSV or PNG.")
```

Pixel Similarity and Network Creation

After loading in libraries and image data, we turned each image into a “network” filled with nodes, or pixels, connected by edges of a certain similarity on the top, bottom, left and right sides. We used two different types of networks, one which included four neighbors, and another which included eight, top, bottom, left, right and all four diagonals.

```python
def create_network(image, sigma, background_pixel, foreground_pixel):
    height, width = image.shape
    n = height * width
    edges = []
```

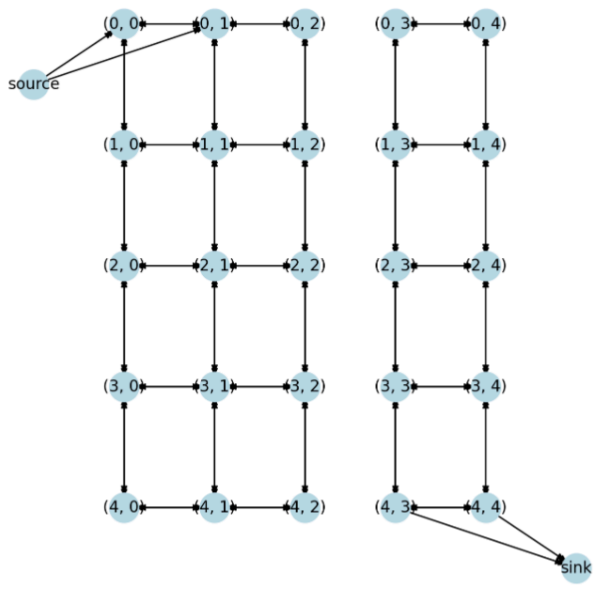
This code chunk defines the network of the image function which determines the similarities of edges between pixels and adds each of these edges to either the source or sink. Pixels of a similar color intensity will have stronger edges. Pixels of opposing or highly differing color will have weaker edges. In grayscale, this value is generally a number from 0 - 255, but as mentioned earlier, was normalized to 0 - 1 in our model. The similarity function which we were provided with calculates the strength of edges based on the numerical similarity between a pixel and its neighboring pixels. Once all the edges between the pixels have been defined as strong, weak or otherwise, we set up the max flow/min cut equation.



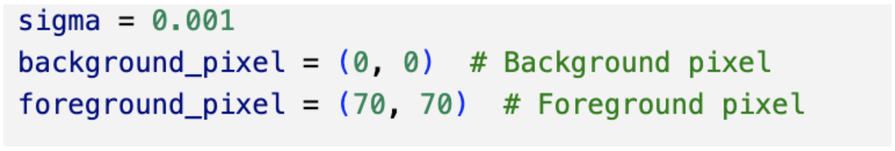
The above graph visually represents the relationship between the source and sink nodes and how they are connected by edges and nodes like the pixels occurring in an image.

Max Flow/Min Cut Theorem and Application

This algorithm is based upon the idea of a "source" or node originating from the background, and a "sink" or node originating from the foreground or object. It is important to select a source and sink which represent characteristic pixels found within the background and foreground because these pixels will be used as "anchors" for which the algorithm maximizes flow among the strongest edges.   We selected (0,0) as our "source" in the top left corner of the image, and (70,70) as our "sink" in the bottom right corner of the image. However, the sink position is adjusted as different images are segmented.



This graph is exactly the same as the previous, except that it contains a minimum cut between the source and sink nodes. This nicely demonstrates how the background has been separated from the foreground through maintaining connections with the source or the sink.

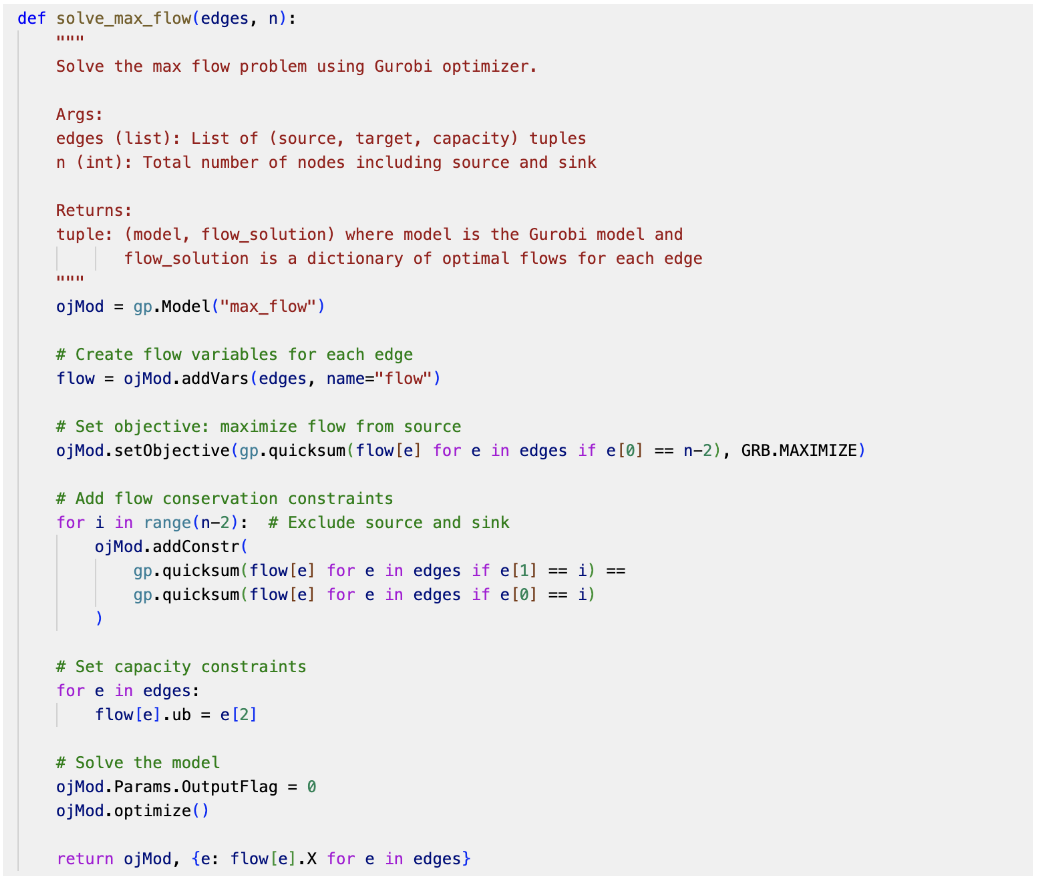


```
sigma = 0.001
background_pixel = (0, 0)  # Background pixel
foreground_pixel = (70, 70)  # Foreground pixel
```

One key idea to note is that selecting the sigma value affects how sensitive your algorithm is to neighboring pixel differences. A high sigma value will decrease the sensitivity and allow for more flow to occur between dissimilar pixel edges, while a low sigma value will increase the sensitivity and allow for less flow to pass through dissimilar pixel edges.

Optimization in Gurobi for Solution Extraction

Now, optimization is utilized to maximize the flow from source to sink. Initially, the flow travels through edges with highest similarity, but when that “flow” hits a dissimilar area (weak edge) it notes that. The algorithm continues to maximize flow everywhere else possible, discovering the entire boundary of weak edges. This set of weak edges, or “minimum cut” represents the boundary between the background and the foreground, where there is the greatest difference in pixel intensity. For each pixel, the amount of flow entering in must be equal to the amount of flow exiting to ensure that “flow” doesn’t get lost inside of the image, except at the source and sink nodes. Furthermore, a capacity constraint was used to create an upper bound for the amount of flow that could be passed through a given edge. For each edge, the maximum capacity is limited to the similarity between neighboring pixels. Once the algorithm runs using these constraints, our solution will display the maximum flow that could occur between two pixels.



```python
def solve_max_flow(edges, n):
    """
    Solve the max flow problem using Gurobi optimizer.

    Args:
    edges (list): List of (source, target, capacity) tuples
    n (int): Total number of nodes including source and sink

    Returns:
    tuple: (model, flow_solution) where model is the Gurobi model and
           flow_solution is a dictionary of optimal flows for each edge
    """
    ojMod = gp.Model("max_flow")

    # Create flow variables for each edge
    flow = ojMod.addVars(edges, name="flow")

    # Set objective: maximize flow from source
    ojMod.setObjective(gp.quicksum(flow[e] for e in edges if e[0] == n-2), GRB.MAXIMIZE)

    # Add flow conservation constraints
    for i in range(n-2):  # Exclude source and sink
        ojMod.addConstr(
            gp.quicksum(flow[e] for e in edges if e[1] == i) ==
            gp.quicksum(flow[e] for e in edges if e[0] == i)
        )

    # Set capacity constraints
    for e in edges:
        flow[e].ub = e[2]

    # Solve the model
    ojMod.Params.OutputFlag = 0
    ojMod.optimize()

    return ojMod, {e: flow[e].X for e in edges}
```

Identifying Cuts Using Depth-First Search

Once the solution from the max flow algorithm was obtained, the next step was to identify the minimum cuts. We used a directed graph to represent the residual graph containing the edges that still have capacity remaining. We performed a depth-first search to identify all of the nodes that are reachable. This allowed us to distinguish between the reachable nodes, which can be accessed from the source, and the non-reachable nodes, which cannot be reached under the current flow configuration. The minimum cut is revealed by the set of edges where one endpoint is in the reachable set of nodes and the other endpoint is in the set of non-reachable nodes. These edges represent the separation of the source from the sink, and what will define the boundary of our image. By using depth-first search, we were able to quickly find the minimum cut by focusing on just the reachable nodes from the source, instead of examining every edge of the graph. This approach speeds up the time the algorithm takes to process the image.

Importantly, the total capacity of these cut edges is the same as the objective value obtained from the max flow algorithm/gurobi optimization.

According to the max-flow min-cut theorem, the maximum flow from the source to the sink in a flow network is equal to or less than the total capacity of the edges that form the minimum cut. Typically, the maximum flow equals the total capacity of the cut. However, when the flow is less than the cut capacity, it may indicate that there are unused edges in the cut, which could result in a suboptimal separation of background and foreground, depending on the coder's interpretation after running the algorithm. A match in these values serves as validation that our cuts are optimal for delineating the foreground from the background image.

```python
def find_cuts(edges, n, flow_solution):
    """
    Find the min cut based on the max flow solution.

    Args:
    edges (list): List of (source, target, capacity) tuples
    n (int): Total number of nodes including source and sink
    flow_solution (dict): Optimal flows for each edge

    Returns:
    list: List of edges representing the min cut
    """
    G = nx.DiGraph()
    for e in edges:
        if e[2] - flow_solution[e] > 1e-6:
            G.add_edge(e[0], e[1])

    reachable = set(nx.dfs_preorder_nodes(G, source=n-2))
    cuts = [(u, v) for (u, v, _) in edges if (u in reachable) != (v in reachable)]

    return cuts
```

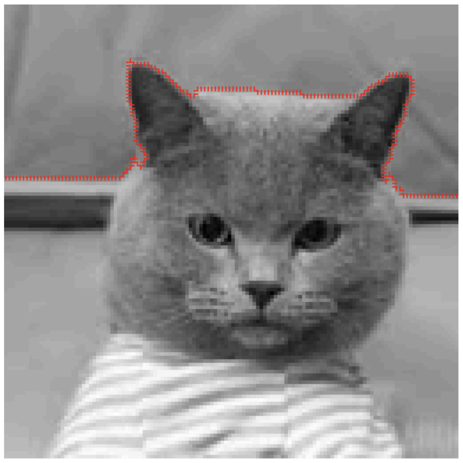
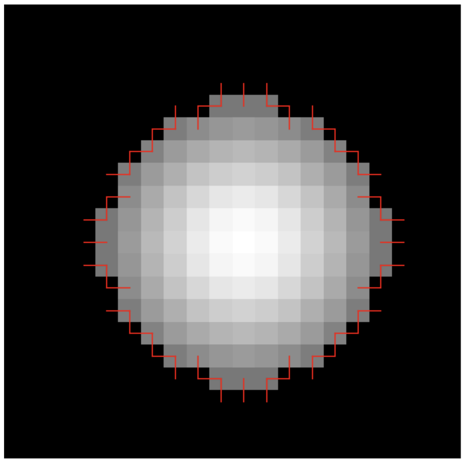
Visualization & Results

Lastly, using the minimum cuts from the maximum flow solution, we are able to display the boundary between the background and foreground of the image. This is done by overlaying the plotted list of cuts onto the grayscale image in red.

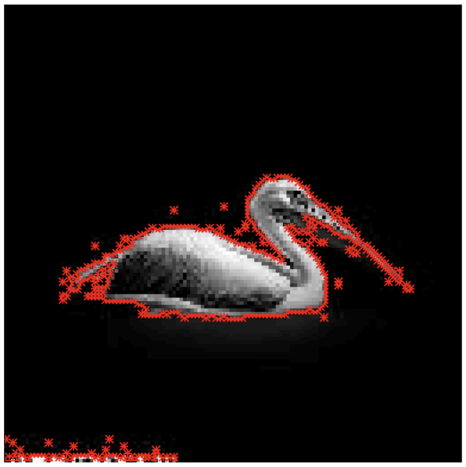
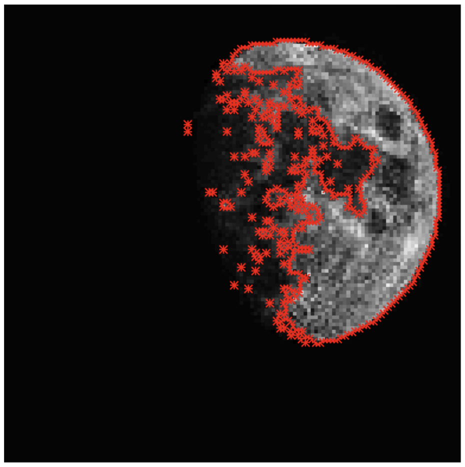
```python
def visualize_results(image, cuts, max_flow):
    """
    Visualize the image segmentation results.

    Args:
    image (numpy.ndarray): Input image
    cuts (list): List of edges representing the min cut
    max_flow (float): Maximum flow value
    """
    height, width = image.shape
    plt.figure(figsize=(10,10))
    plt.imshow(image, cmap='gray')
    for u, v in cuts:
        y1, x1 = divmod(u, width)
        y2, x2 = divmod(v, width)
        plt.plot([x1, x2], [y1, y2], 'r-')
    plt.title(f'Image Segmentation (Max Flow: {max_flow:.2f})')
    plt.axis('off')
    plt.show()
```

And finally, our images reflect the min-cut boundaries for image segmentation:

The foreground and background of the above two images were segmented utilizing the 4 neighboring pixel network to formulate the optimal cuts.

The foreground and background of these above two images were segmented utilizing the 8 neighboring pixel network to formulate the optimal cuts.

Conclusion

By using the max flow/min cut algorithm and optimization techniques, we were able to effectively identify the foreground from the background of an image. The approach used creates a flexible and efficient tool for image segmentation. Furthermore, by utilizing a Gurobi optimization model, we are able to verify the accuracy of our image segmentations through matching the minimum capacity of cut edges to the objective value.

Future Work

In the future, the image segmentation tool could be improved by incorporating an algorithm which dynamically adjusts its own sigma value rather than requiring it to be hard-coded by the user. Additionally, we could expand to support color images and handle images with overlapping objects. Due to the flexible, yet robust framework this tool was built upon, we have the ability to incorporate these improvements into future iterations.

Finally, this project demonstrates an opening for Gurobi as an optimization tool for image segmentation that has a variety of uses and could compete with dated practices such as photoshop and meet the growing need of image segmentation across industries.