Image Segmentation using Minimum cut - Maximum Flow

(Optimization - RM 294)

The University of Texas at Austin
Graduate School

Submitted by:

Alex Parson
Ronak Goyal
Pranav Garg
Sneha Sastry Rayadurgam

Submitted to:

Dr. Daniel Mitchell
Professor

September 2024

Introduction:

Image segmentation is a fundamental problem in the field of image processing, with wide-ranging applications in areas such as object detection, medical imaging, autonomous vehicles, and computer vision. Segmentation involves separating distinct regions within an image, typically isolating the foreground object from its background. In greyscale images, this task becomes a challenge of differentiating between areas of similar intensity levels. Traditional methods often rely on edge detection or thresholding techniques, but these approaches may struggle with noisy or complex images. Thus, more sophisticated techniques are required to accurately partition images for downstream tasks like classification or feature extraction.

In this study, we approach the segmentation problem by framing it as a graph-based optimization problem. Each pixel in the image is represented as a node, and the similarity between neighboring pixel intensities is treated as the edge weight in a network. Our objective is to identify a cut in the graph that minimizes the total weight, effectively separating the background from the foreground. While this appears to be a classic integer programming problem, we leverage the Max Flow/Min Cut Theorem, a well-known result in graph theory, to transform the problem into a linear programming formulation. This theorem allows us to treat the segmentation task as a flow network, where pixel similarities act as flow capacities. By maximizing the flow between known background and foreground pixels, we can efficiently identify the optimal cut for image segmentation.

In our project, we explore the application of the Max Flow/Min Cut Theorem to the segmentation of greyscale images. We implement the algorithm using Python, applying it to a series of test images to evaluate its effectiveness. Our results indicate that the Max Flow/Min Cut approach provides an efficient and robust solution for isolating foreground objects from backgrounds, particularly in complex or noisy images. We also achieved good results on new grayscale images from the web.

Our Goal:

In our startup's initiative to develop a cutting-edge image segmentation tool to compete with established software like Photoshop, we face the challenge of effectively distinguishing between the foreground and background elements of an image. Image segmentation is a crucial task in computer vision that involves partitioning an image into meaningful segments, allowing for easier analysis and manipulation of the image. The goal is to create a system that accurately identifies and separates these segments based on pixel intensity values.

We must first consider how to represent an image as a network. For instance, a typical image can be viewed as a grid of pixel values, such as a 20x20 pixel image containing 400 pixels. Each pixel will be treated as a node in a graph, where the edge (i, j) reflects the similarity between pixel (i) and pixel (j). This graph primarily focuses on neighboring pixels, meaning that only direct connections (e.g., above, below, left and right) will be considered, while non-neighboring pixels will not have direct connections. The similarity between pixels is calculated using an exponential function based on the difference in their intensity values, allowing us to quantify how closely related or connected they are.

In addition to representing pixels as nodes, we introduce two terminal nodes: a source node connected solely to a designated background pixel and a sink node connected only to a chosen foreground pixel.

This structure dictates the flow of information between the source and sink, facilitating the separation of the image into distinct regions.

Minimum Cut - Maximum Flow theorem for Image Segmentation:

The Max Flow/Min Cut Theorem is a key concept in network flow theory that plays a significant role in image segmentation tasks. In this context, an image is modeled as a grid of pixel intensities, where each pixel is represented as a node in a flow network. The connections between these nodes signify the similarities between neighboring pixels, calculated by a similarity measure based on pixel intensities. The objective is to separate the image into distinct regions, typically isolating the foreground object from the background. By utilizing this theorem, we can effectively model the segmentation problem as a flow network, where a source node represents a background pixel and a sink node corresponds to a foreground pixel.

This theorem states that the maximum flow from the source node to the sink node is equal to the minimum cut required to block all flow from the source to the sink. In the image segmentation scenario, maximizing the flow indicates identifying the strongest connections (or similarities) between pixels that contribute to the segmentation of the foreground from the background. Once the maximum flow is established, the residual network is analyzed to identify the minimum cut, which highlights the connections that, when severed, would isolate the foreground from the background. This relationship is critical for determining the optimal cut needed to achieve accurate segmentation.

Some benefits of using this in image segmentation include computational efficiency and enhanced accuracy by converting the segmentation task into a linear programming problem. This method's ability to accurately distinguish between closely related pixel values allows for precise segmentation results, ultimately contributing to the development of a high-quality image segmentation tool for our startup that can compete effectively in the market.

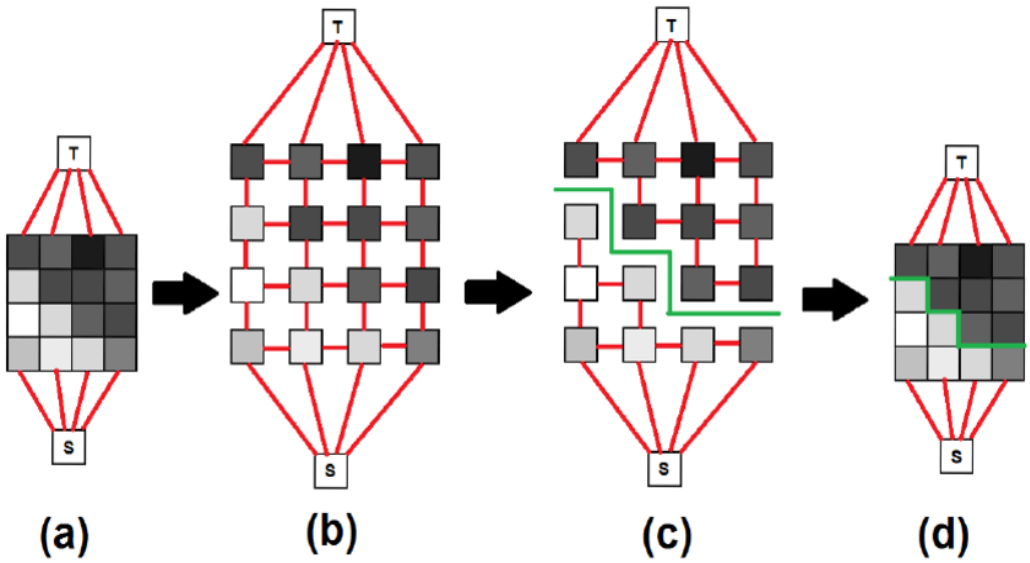


Fig: Step by step illustration of the working of the min cut max flow theorem in image segmentation

Solution:

Summary:

Once the user_input() function is called, the user is prompted to enter all the necessary details like the file path, coordinated of the background and foreground pixel, along with a suitable sigma value, that will be used in calculating the pixel similarity. These inputs are then passed into the main function where the following functions are called to get the final result:

1. **Loading Data**: It calls the load_data() function. This function checks if the input at the file_path is an intensity matrix csv or a .jpg image. If it is a csv, it simply returns that matrix. If it is an image, it creates an intensity matrix and returns that for further processing.
2. **Graph Creation**: The `create_neighbor_similarity_network()` builds a graph representing pixel similarities. It connects adjacent pixels with edges weighted by their similarity. It also designates specific pixels as source and sink nodes. The similarity between 2 pixels is calculated as:

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

   , where I_i is the intensity of pixel i, and Ij is the intensity of pixel j.

3. **Max Flow Optimization**: The `solve_max_flow_undirected()` function formulates a max flow problem using the Gurobi optimizer, maximizing flow from the source to the sink in the graph.
4. **Building Residual Network**: The `build_residual_network_undirected()` constructs a residual graph based on the flow values obtained from the optimization step.
5. **Identifying Accessible Nodes and Min-Cut Edges**: The `get_accessible_nodes()` function finds nodes reachable from the source in the residual network using depth first search . Then, `get_min_cut_edges_undirected()` identifies the edges crossing from the reachable set to the non-reachable set, representing the segmentation boundary.
6. **Visualization**: The code ends by visualizing the segmentation result on the original image using `plot_cuts_on_image()` to display the min-cut edges.

Code snippets and details:

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import gurobipy as gp
from gurobipy import GRB
from PIL import Image
import networkx as nx
```

We start by importing all the necessary packages

```python
def user_input():
    file_path = input("Please enter file path")
    bx = input("Please enter x co-ordinate of background pixel")
    by = input("Please enter y co-ordinate of background pixel")
    fx = input("Please enter x co-ordinate of the foregorund pixel")
    fy = input("Please enter y co-ordinate of the foregound pixel")
    sigma = input("Please enter the sigma value (controls pixel similarity)")
    if bx=='':
        bx=None
    else:
        bx=int(bx)
    if by=='':
        by=None
    else:
        by=int(by)
    if fx=='':
        fx=None
    else:
        fx=int(fx)
    if fy=='':
        fy=None
    else:
        fy=int(fy)
    if sigma=='':
        sigma = 0.05
    else:
        sigma = float(sigma)
    return file_path,bx,by,fx,fy,sigma
```

The **input_data()** function takes the user input. It prompts the user to enter a valid file path, coordinates for background and foreground pixels, and a sigma value that controls pixel similarity. If the values for the coordinates of the background and foreground pixel are not entered, the algorithm then finds the maximum and minimum intensities and then uses those as the values for the background and foreground pixels.

```python
def main(file_path, bx=None, by=None, fx=None, fy=None, sig=10):

    # Step 1: Read the JPEG image and convert it to grayscale
    arr = load_data(file_path)

    # Step 2: Proceed with the rest of the code
    sigma = sig  # Adjust sigma as needed for your image

    G, source_node, sink_node = create_neighbor_similarity_network(arr, sigma, bx, by, fx, fy)

    m, flow = solve_max_flow_undirected(G, source_node, sink_node)


    if m.status == GRB.OPTIMAL:
        flow_values = m.getAttr('x', flow)
        residual_G = build_residual_network_undirected(G, flow_values)
        accessible_nodes = get_accessible_nodes(residual_G, source_node)
        cut_edges = get_min_cut_edges_undirected(G, accessible_nodes)
        plot_cuts_on_image(arr, cut_edges)
    else:
        print("Optimal solution was not found.")
```

The details input by the user are then passed on to the **main()** function where all the necessary steps are run sequentially and if the optimal solution is found by gurobi, it returns the final image with the cuts in red, separating the background from the foreground.

```python
def load_data(file_path):
    if file_path.endswith('.csv'):
        # Load CSV file as intensity matrix and convert to NumPy array
        intensity_matrix = np.loadtxt(file_path, delimiter=',')
        print('its a csv')
    else:
        image = Image.open(file_path).convert('L')
        print('its an image')
        intensity_matrix = np.array(image)
        intensity_matrix = (intensity_matrix / 255.0) * 100

    return intensity_matrix
```

The **load_data()** function checks if the file in the location entered by the user and if the file is a csv having the intensities, then it simply returns that matrix of intensities. If the file is a jpg image , it uses the Image function in the PIL package to convert it into an intensity matrix and returns that matrix for usage in further analysis.

```python
def create_neighbor_similarity_network(arr, sigma, bx, by, fx, fy):
    rows, cols = arr.shape
    G = nx.Graph()
    for r in range(rows):
        for c in range(cols):
            G.add_node((r, c), intensity=arr[r, c])
    neighbors = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    epsilon = 1e-6  # Small constant to prevent zero capacities
    for r in range(rows):
        for c in range(cols):
            for dr, dc in neighbors:
                nr, nc = r + dr, c + dc
                if 0 <= nr < rows and 0 <= nc < cols:
                    I_i = arr[r, c]
                    I_j = arr[nr, nc]
                    similarity = 100 * np.exp(-((I_i - I_j) ** 2) / (2 * sigma ** 2)) + epsilon
                    G.add_edge((r, c), (nr, nc), capacity=similarity)
    if (bx is None) and (by == None) and (fx is None) and (fy == None):
        min_pixel = np.unravel_index(np.argmin(arr), arr.shape) # Replace with appropriate coordinates
        max_pixel = np.unravel_index(np.argmax(arr), arr.shape)      # Replace with appropriate coordinates
        print(f"Background pixel index: {min_pixel}")
        print(f"Foreground pixel index: {max_pixel}")
    else:
        min_pixel = (bx, by)
        max_pixel = (fx, fy)
        print(f"Background pixel index: {min_pixel}")
        print(f"Foreground pixel index: {max_pixel}")
    source_node = 'source'
    sink_node = 'sink'
    G.add_node(source_node)
    G.add_node(sink_node)
    G.add_edge(source_node, min_pixel, capacity=100)
    G.add_edge(max_pixel, sink_node, capacity=100)
    return G, source_node, sink_node
```

The **create_neighbour_similarity_networ()** function takes the user input and starts by creating an empty graph, to which node, or pixels are added along with their intensities. We look at 4 neighbors of every pixel, one right above it, one right below it, one to its left and the last one to its right. We connect every pixel with all 4 of its neighboring pixel described above with an edge that has a weight equal to the similarity between the two pixels, which is given by :

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

, where I_i is the intensity of pixel i, and I_j is the intensity of pixel j. If the user fails to input the coordinates of the foreground and the background pixel, then the next few lines of code find the pixel with min and max intensity and then use that as the background and foreground pixel. We then add a source node (from where the flow starts) and a sink node (where the flow ends) to this graph and connect the background and foreground pixels to the source and sink node respectively with edges having maximum possible weights.

```
def solve_max_flow_undirected(G, source, sink):
    DG = G.to_directed()
    m = gp.Model()
    m.Params.LogToConsole = 0

    flow = m.addVars(G.edges(), lb=0, ub={(u, v): DG[u][v]['capacity'] for u, v in DG.edges()}, name="flow")

    for u, v in G.edges():
        capacity = G[u][v]['capacity']
        m.addConstr(flow[u, v] <= capacity, name=f"cap_{u}_{v}_pos")
        m.addConstr(-flow[u, v] <= capacity, name=f"cap_{u}_{v}_neg")

    net_flow = gp.quicksum(flow[source, neighbor] for neighbor in G.neighbors(source) if (source, neighbor) in flow)
    net_flow -= gp.quicksum(flow[neighbor, source] for neighbor in G.neighbors(source) if (neighbor, source) in flow)
    m.setObjective(net_flow, GRB.MAXIMIZE)

    for node in G.nodes():
        if node == source or node == sink:
            continue
        inflow = gp.quicksum(flow[neighbor, node] for neighbor in G.neighbors(node) if (neighbor, node) in flow)
        outflow = gp.quicksum(flow[node, neighbor] for neighbor in G.neighbors(node) if (node, neighbor) in flow)
        m.addConstr(inflow - outflow == 0, name=f"flow_conservation_{node}")

    m.optimize()
    return m, flow
```

The **solve_max_flow_undirected()** function then takes the graph obtained in the previous function along with the source and the sink nodes and uses gurobi to find the maximin flow. We start off by creating an empty gurobi model and add all the graphs edges as the variables. We add two sets of constraints –

a. Capacity constraints: It ensures the flow from u to v does not exceed the edge's capacity and allows the flow to be negative, which corresponds to the possibility of flow going from v to u. Together, these constraints effectively capture that the absolute flow value should be within the edge's capacity in both directions.
b. Flow Conservation constraints: These constraints ensure that, for every node except the source and sink, the incoming flow equals the outgoing flow, meaning whatever flow enters a node must leave it.

Objective function – to maximize the amount of flow from source to sink. It maximizes the net outgoing flow from the source minus the incoming flow to the source. The function then returns the gurobi model along with the final decision variables. The attributes of this gurobi model are then passed to the next function along with the graph.

```python
def build_residual_network_undirected(G, flow_values):
    residual_G = nx.Graph()
    for u, v in G.edges():
        capacity = G[u][v]['capacity']
        flow_uv = flow_values[u, v]

        residual_capacity_uv = capacity - max(flow_uv, 0)
        residual_capacity_vu = capacity + min(flow_uv, 0)

        if residual_capacity_uv > 1e-6:
            residual_G.add_edge(u, v, capacity=residual_capacity_uv)
        if residual_capacity_vu > 1e-6:
            residual_G.add_edge(v, u, capacity=residual_capacity_vu)
    return residual_G
```

The **build_residual_network_undirected()** function now takes the attributes from the gurobi model and the graph to build a residual network. In flow networks, the residual network represents the additional capacity available for augmenting the flow along each edge. It start of by creating an empty graph that will be filled with edges representing the residual capacities.

It gets the weight for each of the edge as updated capacities that reflect how much additional flow can be pushed in either direction along each edge of the original graph. If there's remaining capacity forward, it adds an edge in that direction. If the flow could potentially be reduced (allowing "reverse flow"), it adds an edge in the opposite direction. The function then returns the residual network graphs for next steps.

```python
def get_accessible_nodes(residual_G, source):
    visited = set()
    stack = [source]
    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            for neighbor in residual_G.neighbors(node):
                if neighbor not in visited:
                    stack.append(neighbor)
    return visited
```

The next function that is called in the solution is the **get_accessible_nodes()** function which performs depth first search by creating a stack to identify the reachable set of nodes using the residual_graph, where edges indicate the residual capacity between , after the maximum flow has been achieved by gurobi. It creates a set – visited, to keep track of nodes that have been visited during the DFS traversal and a stack, to implement the DFS traversal, initialized with the source node. We continue moving as long as there are nodes in the stack, pop a node from the stack and checked if it has already been visited. If it hasn't been visited, it's added to the visited set. All neighbors of this node in the residual graph that haven't been visited are added to the stack for further exploration.

The function then returns the visited set, which contains all nodes that can be reached from the source node in the residual graph. This plays a crucial role in determining which parts of the image should be classified as foreground or background based on reachability in the residual graph.

```python
def get_min_cut_edges_undirected(G, accessible_nodes):
    cut_edges = []
    for u in accessible_nodes:
        for v in G.neighbors(u):
            if v not in accessible_nodes:
                cut_edges.append((u, v))
    return cut_edges
```

We give the output of the DFS traversal – the set of accessible nodes along with the initial graph, to the **get_min_cut_edges_undirected()** function that identifies the edges that form the **minimum cut** in an undirected graph. It starts by creating an empty list that will store the edges forming the minimum cut.

The function iterates over each node u in accessible_nodes and for each node u, it checks its neighbors v in the original graph G. If v is not part of the accessible_nodes, it means there is an edge from a reachable node (u) to an unreachable node (v), which is precisely the definition of a **cut edge**. This pair of nodes is then appended to the cut_edges list. Once each node is traversed, the final cut_edges list is then returned.

```
def plot_cuts_on_image(arr, cut_edges):
    plt.figure(figsize=(4, 4))
    plt.imshow(arr, cmap='gray', interpolation='none')
    ax = plt.gca()
    for u, v in cut_edges:
        if isinstance(u, tuple) and isinstance(v, tuple):
            y1, x1 = u
            y2, x2 = v

            if y1 == y2:
                # Horizontal neighbors, vertical cut between them
                x_cut = (x1 + x2) / 2
                y_cut_start = y1 - 0.5
                y_cut_end = y1 + 0.5
                ax.plot([x_cut, x_cut], [y_cut_start, y_cut_end], color='red', linewidth=1.5)
            elif x1 == x2:
                # Vertical neighbors, horizontal cut between them
                y_cut = (y1 + y2) / 2
                x_cut_start = x1 - 0.5
                x_cut_end = x1 + 0.5
                ax.plot([x_cut_start, x_cut_end], [y_cut, y_cut], color='red', linewidth=1.5)
    plt.title('Min-Cut Edges')
    #plt.gca().invert_yaxis()
    plt.show()
```

The final function – **plot_cuts_on_image()** visualizes the min-cut edges on an image represented by a 2D array given by cut_edges from the get_min_cut_edges_undirected() function and uses the initial intensity matrix to output the final image with cuts. The image array is displayed in grayscale using plt.imshow. The function iterates over each cut_edge pair (u, v), and checks if both u and v are tuples (representing pixel coordinates).Depending on whether u and v are horizontal or vertical neighbors, the function calculates the midpoints for drawing the cuts. If y1 (the y-coordinate of u) equals y2 (the y-coordinate of v), it indicates a horizontal cut. The cut is drawn vertically between the two pixels. If x1 (the x-coordinate of u) equals x2 (the x-coordinate of v), it indicates a vertical cut. The cut is drawn horizontally between the two pixels. The cuts are drawn in red with a specified line width and then plt.show() displays the final plot.

This structured approach allows efficient segmentation of an image by leveraging graph-based optimization and visualization techniques.

Our results on a few training images:

We tested our solution on all the three data sets provided. We started by running the solution on the box csv file, which had the intensities of a 20X20 pixel image as a 20X20 intensity matrix and then ran it for the oval csv – containing the pixel intensities of a 20X20 oval image on a black background. The following results were obtained

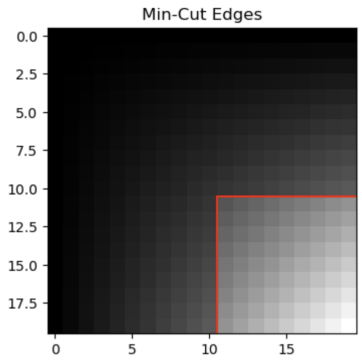


Fig: Result of running the algorithm on box.csv at sigma = 0.004

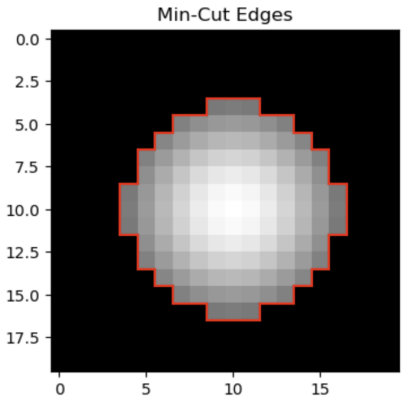


Fig: Result of running the algorithm on oval.csv at sigma = 0.006

We then ran our algorithm to test if it worked well on grayscale images. For this we took the 128X128 pixel grayscale image of a black swan and tested our code. The following results were obtained at sigma value of 0.8.

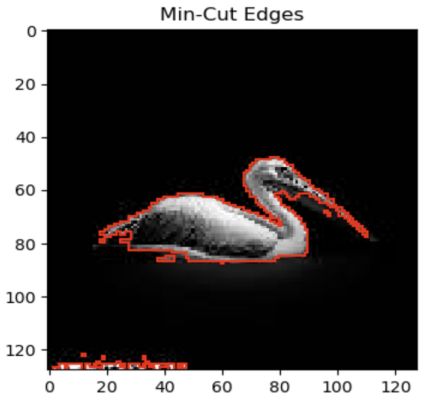
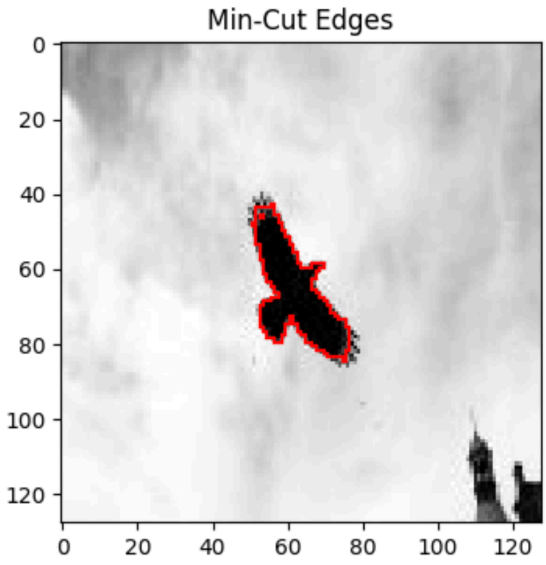


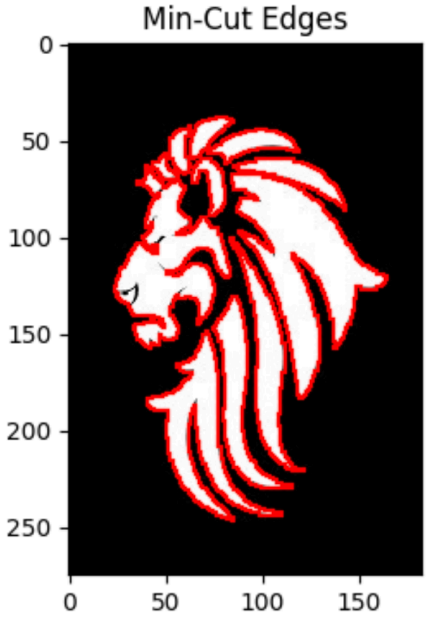
Fig: Result of running the algorithm on swan.jpg at sigma = 0.8

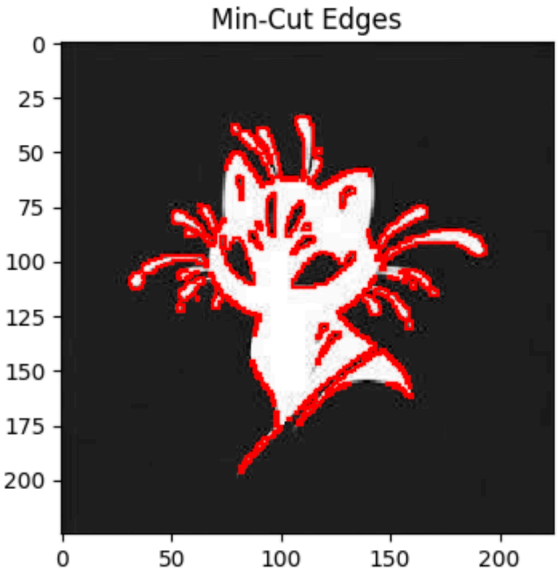
The red cuts made by solution was able to separate the swan from the black background well. We tested this for various values of sigma and found the following cuts at sigma of 0.8

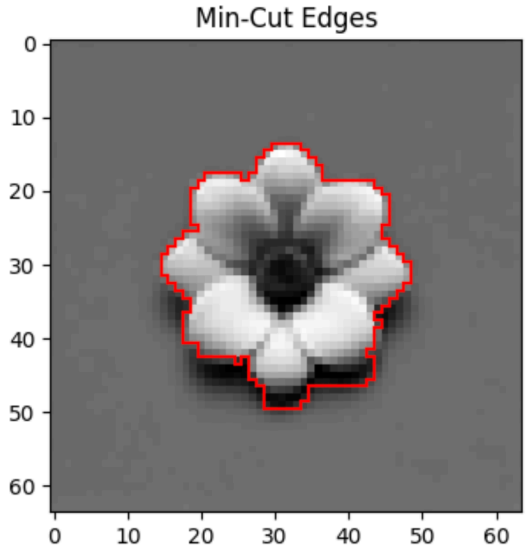
Testing with new images:

To test if our solution worked well for other grayscale images as well, we picked this image from the web and ran our solution. The algorithm was able to identify the edges accurately and make the appropriate cuts to separate the background from the foreground.









Conclusion and Future Scope:

Our solution has demonstrated an effective means of segmenting grayscale images, is capable of recognizing significant changes in pixel intensity and uses this to effectively separate the background from the foreground using red cuts. The results obtained are dependent on the value of sigma used and the time it takes for the final image to be returned depends on the size of the image. The maximum size of the image used by us was 128X128. It would take longer if the image size were increased. Understanding these trade-offs will play a crucial role in exploring methods to scale our solution. The future steps to improve our solution would be to automatically suggest and use the best value of sigma to get the optimal cuts, to scale the code so that it works efficiently even for larger images and color images.

References:

- ResearchGate. (n.d.). Using the max-flow min-cut approach for image segmentation. Retrieved September 28, 2024, from https://www.researchgate.net/figure/Using-the-max-flow-min-cut-approach-for-image-segmentation-a-Define-set-of-pixels-in_fig5_292335453
- Jiang, J. (n.d.). Image segmentation. Retrieved September 28, 2024, from https://julie-jiang.github.io/image-segmentation/
- AI Explained. (2020, August 25). *Image segmentation explained: An intuitive introduction.* YouTube. https://www.youtube.com/watch?v=Tl90tNtKvxs
- AI University. (2020, November 4). *Image segmentation using max flow min cut.* YouTube. https://www.youtube.com/watch?v=7fujbpJ0LB4&t=372s
- OpenAI. (2024). Suggestions from ChatGPT on the code were used based on the logical steps.