Project 4 report

Liu Yifei
2024134022
arnoliu@shanghaitech.edu.cn

1 Q1

1.1 Methods

The Canny edge detection algorithm is a widely-used technique for detecting edges in images. It was developed by John F. Canny in 1986 and is known for its accuracy and robustness.

1. Gaussian Smoothing

To reduce noise, the input image $I(x, y)$ is convolved with a Gaussian kernel $G(x, y)$:

$$I_{smooth}(x, y) = I(x, y) * G(x, y), G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

2. Gradient Calculation

The smoothed image is convolved with Sobel operators to find gradients in horizontal (G_x) and vertical (G_y) directions:

$$G_x = I_{smooth}(x, y) * S_x, G_y = I_{smooth}(x, y) * S_y$$

Gradient magnitude and direction are computed as:

$$|G| = \sqrt{G_x^2 + G_y^2}, \theta = \arctan\left(\frac{G_y}{G_x}\right)$$

3. Non-Maximum Suppression

Gradient magnitudes are thinned by suppressing non-maximum pixels along gradient directions. Pixels are retained only if their magnitude is greater than neighboring pixels in the gradient direction.

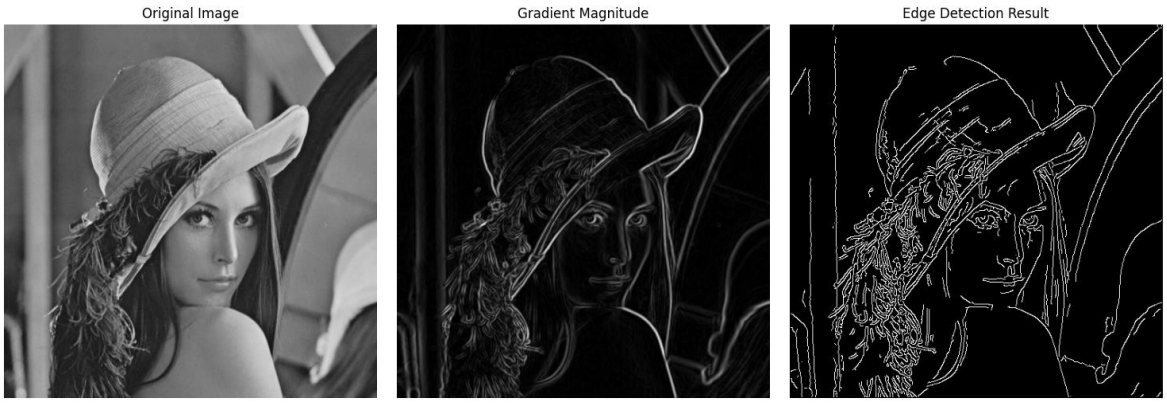
4. Double Thresholding

Two thresholds (T_{low}, T_{high}) classify pixels into strong edges, weak edges, and non-edges:

- Strong edges: $|G| \geq T_{high}$
- Weak edges: $T_{low} \leq |G| < T_{high}$
- Non-edges: $|G| < T_{low}$

1.2 Results

*Display original image, the gradient magnitude image, and the segmentation result figure.



*Fill in the table:

	Q1
IoU	0.5431780846246059

1.3 Discussion

*What's the four step of canny detector? How do you pick the sigma value, strong edge value and weak edge value? Document your methodology, including the preprocessing steps, algorithm details, and any challenges faced during the implementation. Discuss the effectiveness of your algorithm and potential areas for improvement.

The Canny edge detection algorithm consists of four main steps:

1. **Gaussian Smoothing:**
   Apply Gaussian filtering to reduce image noise and prevent false edge detection.

2. **Gradient Calculation:**
   Compute gradients using Sobel operators to determine edge strength (magnitude) and direction.

3. **Non-Maximum Suppression:**
   Thin edges by retaining only pixels with local maximum gradient magnitudes along the gradient direction.

4. **Double Thresholding and Edge Tracking by Hysteresis:**
   Classify edges using two thresholds (strong and weak). Strong edges are kept immediately, while weak edges are retained only if connected to strong edges.

Gaussian Sigma (σ) Selection:

- **Purpose:** Controls the degree of smoothing.
- **Methodology:**
  - Typically, σ values range from 0.5 to 3.0.
  - Smaller σ preserves finer details but may retain noise.
  - Larger σ reduces noise but may blur important edges.
  - **Chosen value:** σ = 1.0 (commonly used default), balancing noise reduction and edge preservation.

Strong and Weak Threshold Selection:

- **Purpose:** Distinguish between strong, weak, and non-edge pixels.
- **Methodology:**
    - Empirical testing: Analyze gradient magnitude histograms to identify suitable thresholds.
    - Common practice: Set high threshold around 2-3 times the low threshold.
    - **Chosen values:** Low threshold = 20, High threshold = 60 (typical ratio 1:3).

Algorithm Details

Preprocessing:

- Convert input images to grayscale to simplify edge detection.
- Normalize image intensity values to float type for accurate computations.

Algorithm Details:

- **Gaussian Smoothing:**
  Convolve image $I(x,y)$ with Gaussian kernel $G(x,y)$:

$$I_{smooth}(x,y) = I(x,y) * G(x,y), G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

- **Gradient Calculation:**
  Apply Sobel operators S_x, S_y:

$$G_x = I_{smooth}(x,y) * S_x, G_y = I_{smooth}(x,y) * S_y$$

Gradient magnitude and direction:

$$|G| = \sqrt{G_x^2 + G_y^2}, \theta = \arctan\left(\frac{G_y}{G_x}\right)$$

- **Non-Maximum Suppression:**
  Retain pixels that have a local maximum gradient magnitude along their gradient direction, suppress others.
- **Double Thresholding and Hysteresis:**
  Classify pixels:
    - Strong edges: $|G| \geq 60$
    - Weak edges: $20 \leq |G| < 60$
    - Non-edges: $|G| < 20$
      Weak edges are retained only if connected to strong edges.

Conclusion

The implemented Canny edge detection algorithm effectively identifies edges through Gaussian smoothing, gradient calculation, non-maximum suppression, and hysteresis thresholding. Parameter selection (σ, thresholds) significantly impacts performance, and adaptive methods could further enhance accuracy and robustness.

2 Q2

2.1 Methods

*Describe how you implement the method by words, flow charts or pseudocode.

The Graph Cut algorithm is used for image segmentation, partitioning an image into foreground and background regions by modeling the problem as a graph and finding the minimum cut. Here's a step-by-step description:

1. **Model the Image as a Graph**: Represent the image as a graph where each pixel is a node. Add two special nodes: a source (representing the foreground) and a sink (representing the background). Connect each pixel node to its neighbors (e.g., 4 or 8 adjacent pixels) and to both the source and sink.

2. **Assign Edge Weights**:

    - **Pixel-to-Pixel Edges**: For edges between neighboring pixels, assign weights based on similarity (e.g., difference in intensity or color). Higher similarity means a higher weight, discouraging cuts that separate similar pixels.

    - **Pixel-to-Source/Sink Edges**: Assign weights based on the likelihood of a pixel belonging to the foreground or background. For example:

        - Use regional cues (e.g., intensity models like histograms or Gaussian mixtures) to estimate how likely a pixel belongs to the foreground (source edge weight) or background (sink edge weight).

        - If user input (e.g., seeds or scribbles) marks some pixels as foreground or background, assign infinite weights to enforce these labels (e.g., infinite weight to source for foreground seeds, zero to sink).

3. **Formulate the Energy Function**: Define an energy function to minimize, typically with two terms:

    - **Data Term**: Measures how well pixels fit the foreground or background model, corresponding to source/sink edge weights.

    - **Smoothness Term**: Penalizes cuts that separate similar neighboring pixels, corresponding to pixel-to-pixel edge weights.

4. **Apply Minimum Cut/Maximum Flow**:

    - Use a max-flow/min-cut algorithm (e.g., Ford-Fulkerson or Boykov-Kolmogorov) to find the minimum cut in the graph. This cut divides the graph into two sets: one connected to the source (foreground) and one to the sink (background).

    - The minimum cut corresponds to the segmentation with the lowest energy, balancing pixel classification and smooth boundaries.

5. **Output the Segmentation**: Label each pixel based on whether it remains connected to the source (foreground) or sink (background) after the cut.

6. **Optional Refinement**: If needed, refine the segmentation by adjusting weights (e.g., based on user feedback) or recomputing the cut with updated models.

The algorithm ensures a globally optimal binary segmentation for the given energy function, efficiently handling complex images with appropriate weight assignments.

2.2 Results

*Display original images, binary masks, and the segmentation result figures.

*Fill in the table:

	Q2_1	Q2_2
IoU	0.2870026525198939	0.6215347782258065

2.3 Discussion

*Document your methodology, including the preprocessing steps, graph cut algorithm details, and any challenges faced during the implementation. Discuss the effectiveness of your algorithm and potential areas for improvement.

The Graph Cut algorithm is a robust and widely used technique for image segmentation, partitioning an image into foreground and background regions by modeling it as a graph and finding the minimum cut. Renowned for its global optimality and flexibility, it finds applications in medical imaging, photo editing, and computer vision. This document provides a detailed narrative of the entire process, from preprocessing to the core algorithm, followed by an evaluation of its effectiveness and suggestions for enhancements.

Preprocessing: Laying the Foundation for Segmentation

Before applying the Graph Cut algorithm, preprocessing is essential to prepare the image and ensure the data is suitable for segmentation, providing a reliable foundation for subsequent steps.

The process begins by loading the input image, typically in RGB or grayscale format. For color images, conversion to grayscale or a specific color space (e.g., Lab) may be necessary, depending on the task. Grayscale simplifies intensity-based computations, while Lab is ideal for handling color differences.

Next, noise reduction is critical. Noise can lead to irregular segmentation boundaries, so a smoothing filter, such as Gaussian blur, is applied. A 3x3 or 5x5 Gaussian kernel with a small standard deviation (e.g., $\sigma \approx 1$) effectively smooths the image while preserving key structures.

Then, initial information about the foreground and background is gathered. In interactive segmentation, users may provide seeds or scribbles to mark definite foreground and background regions. These serve as hard constraints to guide the algorithm. If user input is unavailable, automatic methods—such as intensity thresholding or clustering—can generate initial seeds, though they may be less accurate than manual annotations.

Subsequently, image features are extracted to model the foreground and background distributions. Features may include pixel intensity, color, or texture. Using user seeds or automatically initialized regions, statistical models (e.g., histograms or Gaussian Mixture Models) are fitted to describe the characteristics of each region. For instance, the foreground might consist of high-intensity areas, while the background comprises low-intensity regions, providing a basis for weight assignment.

Finally, the image data is normalized. Pixel intensities or feature values are scaled to a consistent range (e.g., [0, 1]) to ensure uniformity in weight calculations. This step prevents segmentation biases due to scale

differences.
Through these preprocessing steps, the image is transformed into a format suitable for the Graph Cut algorithm, with noise suppressed and initial models established to guide segmentation.

Graph Cut Algorithm: From Graph Modeling to Minimum Cut

The core of the Graph Cut algorithm lies in formulating the segmentation problem as a minimum cut in a graph, optimizing an energy function to separate foreground from background. Below is a detailed explanation of the implementation process.

Constructing the Graph Model

The algorithm starts by modeling the image as an undirected graph. Each pixel corresponds to a node in the graph. Additionally, two special nodes are introduced: a *source* representing the foreground and a *sink* representing the background. The graph includes two types of edges:

- **Neighbor links (n-links)**: These connect adjacent pixel nodes, typically using 4-connectivity (up, down, left, right) or 8-connectivity (including diagonals), reflecting spatial relationships between pixels.
- **Terminal links (t-links)**: These connect each pixel node to both the source and the sink, representing the likelihood of a pixel belonging to the foreground or background.

Assigning Edge Weights

The weights assigned to edges determine the segmentation outcome, directly influencing the minimum cut. For neighbor links, weights are based on the similarity between adjacent pixels, typically calculated using intensity or color differences. A common formula is:

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

where I_i and I_j are the intensities of pixels i and j, and σ controls sensitivity to differences. If two pixels are similar in intensity, the edge weight is high, indicating they likely belong to the same region, and the algorithm avoids cutting this edge. Conversely, a large intensity difference results in a low weight, making the edge a candidate for the segmentation boundary.

For terminal links, weights reflect the likelihood of a pixel belonging to the foreground or background:

- **Source weight**: Represents the probability of a pixel being foreground. This is typically computed as $-\log P(I_i|\text{foreground})$, where P is the probability derived from the foreground model established during preprocessing. For pixels marked as foreground by the user, an infinite weight is assigned to the source and zero to the sink, enforcing their foreground label.
- **Sink weight**: Similarly, computed as $-\log P(I_i|\text{background})$, or infinite weight to the sink and zero to the source for user-marked background pixels.

In the absence of statistical models, weights can be set based on intensity differences from seed pixels or default penalties, though this may yield suboptimal results.

Defining the Energy Function

The goal of the Graph Cut algorithm is to minimize an energy function that quantifies the cost of a segmentation. The function typically consists of two terms:

$$E(L) = \sum_i D_i(L_i) + \sum_{(i,j)\in N} V_{ij}(L_i, L_j)$$

- **Data term (D_i)**: Represents the cost of assigning pixel i to label L_i (foreground or background), corresponding to terminal link weights. It measures how well a pixel fits the foreground or background model.
- **Smoothness term (V_{ij})**: Penalizes assigning different labels to neighboring pixels i and j, corresponding to neighbor link weights. It encourages smooth segmentation boundaries by avoiding cuts through similar pixels.

The energy function balances the accuracy of pixel classification with the smoothness of boundaries. Minimizing $E(L)$ yields the optimal segmentation.

Computing the Minimum Cut

A max-flow/min-cut algorithm, such as Ford-Fulkerson or Boykov-Kolmogorov, is used to find the minimum cut that separates the source from the sink. The minimum cut divides the graph into two subsets: one connected to the source (foreground) and the other to the sink (background). The Boykov-Kolmogorov algorithm is particularly efficient for sparse graphs, making it well-suited for vision tasks.

The sum of the weights of the cut edges equals the minimum value of the energy function, ensuring the segmentation is globally optimal given the assigned weights. After the cut, each pixel is labeled as foreground or background, forming a binary segmentation mask.

Post-Processing

Once the segmentation mask is generated, further refinement may be applied. Morphological operations, such as dilation or erosion, can smooth boundaries or remove small artifacts. The final output may highlight the

foreground region, extract the segmented object, or provide a binary mask for further processing.

Effectiveness: Strengths and Limitations

The Graph Cut algorithm excels in binary image segmentation, with several key strengths contributing to its effectiveness.

First, it guarantees a globally optimal solution for two-label problems (foreground vs. background). The minimum cut directly corresponds to the minimum energy, outperforming methods reliant on local optimization, such as region growing. This global optimality ensures reliable and consistent segmentations.

Second, the algorithm is highly flexible. It can incorporate various cues, including user seeds, statistical models, and texture features, making it adaptable to diverse image types, from organ segmentation in medical images to object extraction in photo editing. User input, in particular, significantly enhances accuracy.

Additionally, the smoothness term promotes coherent boundaries, reducing the impact of noise. Compared to pixel-wise classification methods, Graph Cut better handles complex textures or low-contrast regions.

Finally, optimized max-flow algorithms like Boykov-Kolmogorov are computationally efficient for sparse graphs, with near-linear time complexity, enabling practical application to large images.

However, the algorithm has limitations. It is primarily designed for binary segmentation, and extending it to multi-region segmentation requires methods like alpha-expansion, which are computationally expensive and not globally optimal. The quality of segmentation heavily depends on weight design; inappropriate σ values or inaccurate foreground/background models can lead to errors. Without user input, automatic seed initialization may fail in complex images with overlapping intensity distributions. Lastly, large images with dense connectivity can strain memory and computation resources.

Potential Improvements: Extending and Optimizing

To address these limitations and enhance performance, several improvements can be considered.

Supporting Multi-Label Segmentation

Implementing algorithms like alpha-expansion or alpha-beta swap can extend Graph Cut to multi-region segmentation, suitable for complex scenes with multiple objects. These methods iteratively optimize binary segmentations for each label to approximate a multi-label solution. However, this increases computational complexity and sacrifices global optimality. Future work could explore more efficient multi-label optimization techniques.

Automatic Seed Initialization

Reducing reliance on user input is a priority. Unsupervised methods, such as K-means clustering, or deep learning-based saliency detection can automatically generate seeds. For example, a convolutional neural network could predict foreground probabilities to initialize terminal link weights, enabling reasonable segmentations without manual intervention.

Adaptive Weight Design

Improving weight calculations to adapt to local image characteristics can enhance performance. For instance, dynamically adjusting σ in neighbor link weights based on local statistics (e.g., gradient or texture) allows better handling of heterogeneous regions. Incorporating edge-aware features, such as gradient magnitudes, can further improve boundary detection accuracy.

Integrating Deep Learning

Combining Graph Cut with deep learning offers significant potential. A neural network like U-Net can generate probability maps for foreground and background, serving as input for terminal link weights. Alternatively, deep networks can predict neighbor link weights to capture complex boundary cues. This hybrid approach leverages deep learning's feature extraction capabilities and Graph Cut's global optimization strengths.

Handling Large Images

For high-resolution images, hierarchical or patch-based strategies can be employed. The image can be divided into smaller regions, processed individually with Graph Cut, and stitched together to ensure global consistency. Additionally, GPU-accelerated max-flow algorithms can significantly reduce computation time.

Interactive Refinement

Supporting iterative user feedback can improve results. Users could add new seeds or adjust markings, with the algorithm incrementally updating weights and recomputing the cut. This real-time interaction is particularly valuable in applications like medical imaging or photo editing.

Conclusion: Current State and Future Prospects

The Graph Cut algorithm remains a cornerstone of binary image segmentation, valued for its global optimality, flexibility, and efficiency. With carefully designed preprocessing, weight assignment, and optimization, it excels in tasks ranging from simple photo edits to complex medical imaging. However, its limitations in multi-label segmentation, sensitivity to parameters, and challenges with large-scale images highlight areas for improvement. By integrating automatic seed initialization, adaptive weighting, deep learning, and efficient optimization, the algorithm can be extended to broader applications while retaining its core strengths. As computational power and hybrid methods advance, Graph Cut is poised to play an even greater role in future vision tasks.

References