SRH University Heidelberg

Task 2:
Image Segmentation Using Graph-Based Technique in Computer Vision

Author:	Nikhil Kumar
Matriculation Number:	11037894
Email Address:	nikhil.kumar@stud.hochschule-heidelberg.de
Date of Submission:	12th June 2024

Under the Guidance of Professor
Dr. Ing Milan Gnjatovic

Aim of the Task:

This project aims to develop a program that implements a graph-based image segmentation algorithm. The goal is to partition an image into meaningful regions by leveraging graph theory, where each pixel represents a node and edges between nodes represent the similarity or dissimilarity between pixels based on intensity differences. This will be achieved by creating and processing edges between adjacent pixels, sorting them by weight, and merging segments based on a defined threshold. The implementation will be evaluated for its accuracy, efficiency, and robustness in handling various types of images. Ultimately, the project aims to enhance image analysis and processing by visually representing the segmented regions and contributing to advancements in computer vision and image processing applications.

Objective of the Task :

The objective of this project is to implement a graph-based image segmentation algorithm, which is designed to partition an image into meaningful regions for further analysis. This algorithm utilizes the principles of graph theory, where pixels or groups of pixels are represented as nodes, and edges between nodes denote the similarity or dissimilarity between them based on various image features such as color, intensity, or texture. The primary goal is to achieve an accurate segmentation that aligns with the perceptual organization of objects within the image, facilitating tasks like object recognition, tracking, and image understanding. By developing this algorithm, the project aims to enhance image processing capabilities, enabling more precise and efficient segmentation for a wide range of applications in computer vision, medical imaging, and multimedia.

Introduction :

Image Segmentation has long been an interesting problem in the field of image processing as well as object detection. The problem of segmenting an image into regions could directly benefit a wide range of computer vision problems, given that the segmentations were reliable and efficiently computed. [1a]

For example, image recognition can make use of segmentation results in matching to address problems such as figure-background separation and recognition by parts.[1a]

To solve this segmentation problem, an Image segmentation approach called the Graph-based Image segmentation algorithm was proposed by Felzenszwalb and Huttenlocher.

(Note: In this task, a simplified version of the Felzenszwalb and Huttenlocher Image segmentation algorithm is implemented where the threshold value is constant i.e. the threshold value is an input parameter.)

Graph-based image segmentation, introduced by Felzenszwalb and Huttenlocher, is a seminal method in computer vision that aims to divide an image into cohesive regions by leveraging the principles of graph theory. Within this framework, pixels are represented as nodes in a graph, and edges between nodes encode the similarity or dissimilarity between adjacent pixels. Notably, the algorithm incorporates a constant threshold value to guide the segmentation process, ensuring that segments are merged based on predefined criteria. By iteratively constructing and analysing edges, sorting them by weight, and merging segments that satisfy the threshold condition, the algorithm efficiently generates meaningful partitions of the image. Renowned for its effectiveness across diverse image datasets and its computational efficiency, this approach has become a cornerstone in image segmentation, offering valuable insights for various applications in computer vision research and image processing tasks.

What is Image Segmentation?

Image segmentation is a fundamental process in computer vision and image processing that involves the process of dividing an image into multiple parts or segments to simplify and/or change the representation into something more meaningful and easier to analyze. [1b]

Here are some key aspects of image segmentation:

1. **Purpose:** The primary aim of image segmentation is to identify and isolate objects and boundaries within an image. This can facilitate various tasks such as object recognition, tracking, image editing, and medical image analysis.

2. **Methods**: There are various techniques for image segmentation, including thresholding, edge detection, region-based segmentation, clustering, and graph-based methods. Each method has its advantages and is chosen based on the specific requirements of the task.

Applications of Image Segmentation:

- **Medical Imaging:** To identify and isolate regions of interest such as tumors, organs, or tissues.

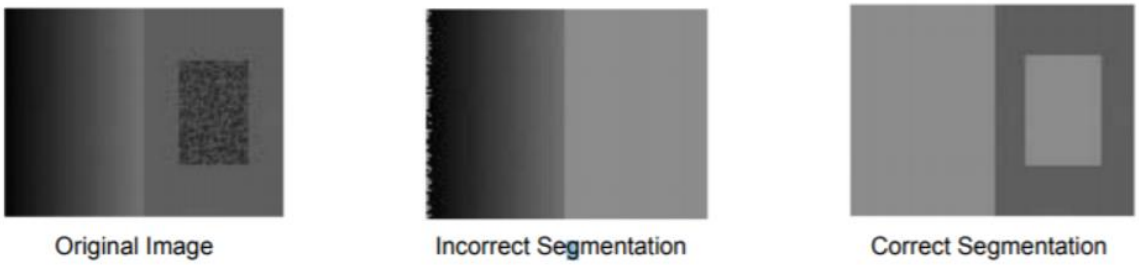
- **Object Detection and Recognition:** To detect and recognize objects within an image for applications in robotics, autonomous driving, and security systems.

- **Image Compression and Editing:** To segment and manipulate parts of an image for better compression or for tasks like background removal and object replacement.

What is the role of image segmentation in image processing?

Image segmentation plays a pivotal role in image processing by dividing an image into meaningful regions, making it easier to analyze and interpret its content. By isolating objects and delineating boundaries within an image, segmentation aids in object recognition, boundary detection, and image feature extraction, which are vital for further image analysis or object tracking. It facilitates various higher-level tasks such as object recognition, classification, and tracking by simplifying complex images into segments that share similar characteristics like color, intensity, or texture. This process reduces computational complexity and enhances the efficiency of subsequent processing steps, making segmentation an essential component in fields such as medical imaging, autonomous driving, and more.[1b]

Below is a visual demonstrating image segmentation



[1a]

Why image segmentation is important?

Image segmentation is crucial in image processing because it simplifies the analysis and interpretation of images by dividing them into meaningful regions. This segmentation allows for the isolation of objects and delineation of boundaries within an image, facilitating higher-level tasks such as object recognition, classification, and tracking. By breaking down complex images into simpler segments that share similar characteristics like color, intensity, or texture, segmentation reduces computational complexity and enhances the efficiency of subsequent processing steps. This is particularly important in fields like medical imaging, where precise identification of tissues and abnormalities can significantly impact diagnostic accuracy and treatment planning.

Furthermore, image segmentation plays a vital role in applications requiring real-time processing, such as autonomous driving and security surveillance. In these scenarios, the ability to quickly and accurately segment images into distinct regions allows for reliable object detection and tracking, contributing to safer navigation and more effective monitoring. Additionally, segmentation aids in feature extraction by focusing on specific regions of interest, thereby improving the accuracy of tasks like pattern recognition and image classification. Overall, image segmentation is an essential process that underpins various advanced applications, making it a cornerstone of modern image processing and analysis.

Where in the real world is image segmentation used?

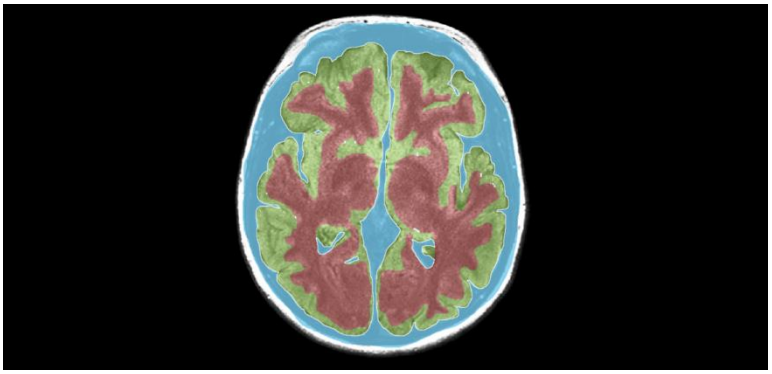
Since there are numerous real-world use cases, I've listed the most significant and intriguing ones below.

Autonomous Vehicles:

[1c]

Image and video segmentation is critical for training robust computer vision models to enable autonomous driving capabilities. Segmentation provides precise delineation of objects like vehicles, pedestrians, roads, and traffic signs. This pixel-level understanding feeds accurate perception algorithms for tasks like obstacle avoidance, lane keeping, and navigation. Segmented data also trains models to segment and interpret new driving scenes. Reliable autonomy requires extensive, diverse driving imagery with meticulous semantic, instance, and panoptic segmentation masks labeling all environmental elements. This high-quality training data produces self-driving models that precisely perceive complex, dynamic driving environments. [1c]

Medical Imaging Analysis:



[1c]

- **Tumor Detection**: Segmentation helps in identifying and isolating tumors and other abnormal growths in medical scans such as MRI, CT, and PET images. AI models trained on segmented medical imaging datasets are revolutionizing healthcare diagnostics and treatment planning. Segmentation allows isolating and measuring specific anatomy in modalities like CT, MRI, and ultrasound. For example, organ, tumor, and tissue segmentation quantify volumes and morphology. Segmentation also aids visualization, surgery planning, and pathology detection by highlighting

anatomical structures. Robust AI models for medical image segmentation require training data with diverse scanned images expertly annotated at the pixel level. This powers the precision segmentation of novel scans to augment workflows. Overall, medical segmentation enables quantitative, automated analysis of anatomical imagery impossible through human inspection alone. [1c]

- **Organ Segmentation**: It assists in delineating organs and structures within the body, which is crucial for surgical planning, diagnosis, and treatment monitoring.

- **Cell Counting and Analysis**: In pathology, segmentation is used to count and analyze cells in microscopic images, aiding in disease diagnosis and research.

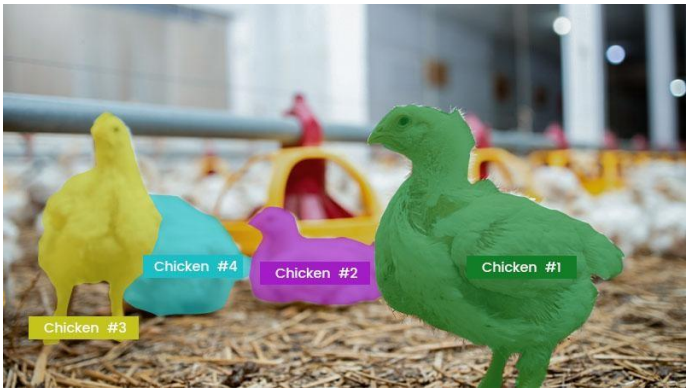
Analysis of Satellite Images:

[1c]

Land Use and Land Cover Classification: Segmentation is used to classify different land cover types (e.g., forests, urban areas, water bodies) in satellite images, aiding in environmental monitoring and urban planning.

Also, AI models trained with segmentation datasets can identify various geographic and man-made features with high precision. They help in tasks like land cover classification, urban planning, environmental monitoring, and disaster management. For instance, segmentation models can differentiate forests, water bodies, urban areas, and agricultural lands in a satellite image, providing valuable data for land use planning and climate change studies. Image segmentation algorithms can also identify damaged structures or flooded areas in post-disaster satellite images, aiding in efficient disaster response. [1c]

Smart Agriculture:



[1c]

In agriculture, segmentation of aerial imagery provides intelligence on crop health and land use. Segmenting fields, soil, irrigation, and crops in satellite or drone photographs enables advanced farm analytics. Computer vision models can automate plant counting, assess ripeness and disease, monitor growth, and estimate yields by leveraging precise segmentation. Image segmentation dataset allows the development of highly accurate models even for niche crops. Segmentation thereby unlocks automation to optimize farming operations, catch issues early, and improve sustainability.[1c]

Industrial Inspection:

[1c]

Quality Control: Segmentation is used in automated inspection systems to detect defects or irregularities in manufactured products, ensuring quality and consistency.
Robotic Vision: It enables robots to recognize and manipulate objects in complex environments, facilitating tasks such as sorting and assembly.

Also, they can identify and localize defects in products or components based on images or video feeds, reducing reliance on manual inspection. For instance, in electronics manufacturing, segmentation models can detect faulty components on a circuit board. In the automotive industry, they can identify dents or scratches on vehicle bodies. These applications not only improve inspection accuracy and speed but also reduce costs and enhance product quality. [1c]

Types of Image Segmentation:

1. Edge-Based Segmentation

Edge-based segmentation involves detecting edges or pixels between different regions. The condition for different regions may be a rapid transition of intensity. So those pixels are extracted and linked together to form a closed boundary. This technique is useful for images with clear boundaries between objects and backgrounds.
Edge-based segmentation is widely used in computer vision applications, such as object recognition and tracking. It is also used in medical image processing for identifying anatomical structures.[1b]

2. Morphological-Based Segmentation

Morphological-based segmentation involves using mathematical morphology to extract features from an image. This technique is useful for images with irregular shapes and structures. Morphological-based segmentation involves operations such as dilation, erosion, opening, and closing to extract features from an image.
Morphological-based segmentation is widely used in medical image processing for identifying anatomical structures. It is also used in traffic analysis for vehicle detection and tracking. [1b]

3. Threshold-Based Segmentation

Threshold-based segmentation is one of the simplest and most commonly used techniques for image segmentation. It involves setting a threshold value and classifying pixels based on their intensity values. Pixels with intensity values above the threshold are classified as one segment, while those below the threshold are classified as another segment.
This technique is useful for images with clear boundaries between objects and backgrounds. Threshold-based segmentation is widely used in medical image processing, such as identifying tumors and other abnormalities. It is also used in traffic analysis for vehicle detection and tracking. [1b]

4. Bayesian-Based Segmentation

Bayesian-based segmentation involves using probability to construct models based on the image data. This technique involves estimating the probability distribution of the image data and using it to segment the image. Bayesian-based segmentation is useful for images with complex structures and textures.
Bayesian-based segmentation is widely used in various fields, including healthcare, traffic analysis, and pattern recognition. In healthcare, it is used for medical image processing, such as identifying tumors and other abnormalities. In traffic analysis, it is used for vehicle detection and tracking. In pattern recognition, it is used for object recognition and classification. [1b]

5. Neural Network-Based Segmentation

Neural network-based segmentation involves using artificial neural networks to segment images. This technique involves training a neural network to recognize patterns in the image data and using it to segment the image. Neural network-based segmentation is useful for images with complex structures and textures.
Neural network-based segmentation is widely used in various fields, including healthcare, traffic analysis, and pattern recognition. In healthcare, it is used for medical image processing, such as identifying tumors and other abnormalities. In traffic analysis, it is used for vehicle detection and tracking. In pattern recognition, it is used for object recognition and classification. [1b]

Graph-Based Image Segmentation:

Graph-based image segmentation is a technique that leverages the principles of graph theory to partition an image into meaningful regions. This method treats an image as a graph, where each pixel or group of pixels is represented as a node, and the weight of an edge between nodes encodes the similarity or dissimilarity based on certain criteria such as color, intensity, or texture.
The main objective of GBS is to divide an image into separate regions, each one representing a segment in the image.[1f]

The figure below provides an example of (**a) the input image (b) the same image after simple segmentation**. Each segment is assigned a randomly selected color.[1e]

[1e]

A Graph-Based Image Segmentation Algorithm

In this approach, an image is represented as an undirected graph G = (V, E)
Where,

- each pixel (x_i, y_i) is represented by a node $n_i \in V$,
- set of edges E contains all pair of neighboring pixels, i.e.,

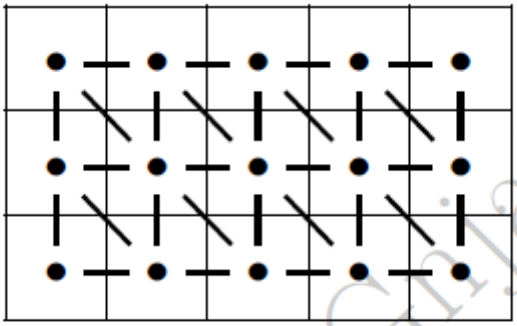
$$(n_i, n_j) = ((x_i, y_i), (x_j, y_j)) \in E \Leftrightarrow (x_i = x_j \wedge y_i + 1 = y_j) \vee (x_i + 1 = x_j \wedge y_i = y_j) \vee (x_i + 1 = x_j \wedge y_i + 1 = y_j)$$

[1e]

- each edge $((x_i,y_j),(x_i,y_j)) \in E$ is assigned a weight that is equal to absolute intensity difference between the pixels, i.e.,

$$d(n_i,n_j) = d((x_i,y_j),(x_j,y_j)) = |f(x_i,y_i) - f(x_j,y_j))|\,, \qquad (6.2)$$

where $f(x,y)$ represents the intensity of image f at pixel (x,y). [1e]



[1e]

An illustration of an undirected graph representing an image is shown above

Steps to Perform Graph-Based Image Segmentation Algorithm

Step 1: Assigning each pixel to its own Segment

Throughout the algorithm execution, current segmentation results are represented by integer array: [1e]

$$\mathscr{C} = (c(n_1), c(n_2), \ldots, c(n_k)) \qquad [1e]$$

Where $(\forall\, 1 \le i \le k)(c(n_i) \in \{1,2,\ldots,k\})$, and $c(n_i)$ represents the identification number of the segment to which the pixel $\mathbf{n_i}$ is currently assigned. In this Step, each pixel is assigned to its own segment, i.e. [1e]

$$(\forall\, 1 \le i \le k)(c(n_i) = i)\,. \qquad [1e]$$

Example: For a 3x3 image, the segments array will be: **[0, 1, 2, 3, 4, 5, 6, 7, 8].**

Step 2: Edge Sorting

The set of edges E is ordered by non-decreasing edge weight:

$$\hat{E} = (n_{i_1}, n_{j_1}), (n_{i_2}, n_{j_2}), \ldots, (n_{i_m}, n_{j_m}) \quad \text{[1e]}$$

- Sort all edges by their weights (intensity differences).

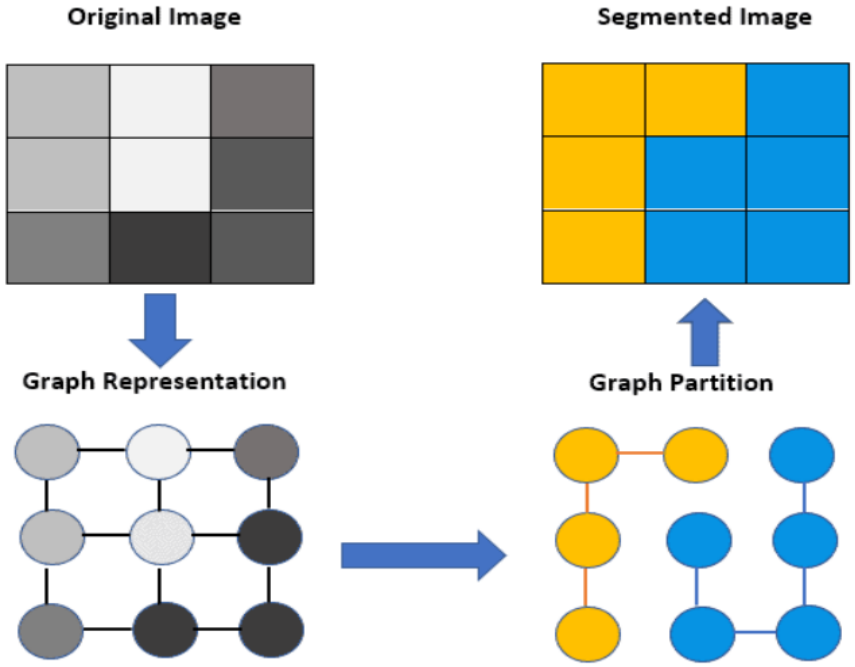
Example: For a small part of a 3x3 image with edges having weights [0, 10, 30], the sorted edges will be: [0, 10, 30].

Step 3: Segmentation and Merging

We iterate through sequence E from the first to the last position. For each edge $(n_i, n_j) \in E$, if pixels n_i and n_j belong to different segments that is if $c(n_i) \neq c(n_j)$ and the weight $d(n_i, n_j)$ is less than or equal to a given threshold value **t** then segments **c(ni)** and $c(n_i)$ are merged. [1e]

$$\begin{aligned} &for\ (1 \le p \le |\hat{E}|)\ \{ \\ &\quad if\ ((c(n_{i_p}) \neq c(n_{j_p})) \wedge (d(n_{i_p}, n_{j_p}) \le \tau))\ then \\ &\quad\quad for\ (1 \le q \le |\mathscr{C}|) \\ &\quad\quad\quad if\ (c(n_q) = c(n_{j_p}))\ then\ (c(n_q) \leftarrow c(n_{i_p})) \\ &\} \end{aligned} \quad \text{[1e]}$$

Below is an illustration of how an original image is transformed into a segmented image



[1f]

What is the reason for choosing the Graph-Based technique as an image segmentation technique?

The algorithm runs in time nearly linear in the number of graph edges and is also fast in practice. An important characteristic of the method is its ability to preserve detail in low-variability image regions while ignoring detail in high-variability regions.[1g]

The above statement refers to the selective and adaptive nature of the graph-based image segmentation algorithm.
Here's a detailed explanation of what this means:

Low-Variability Image Regions

- **Definition**: Low-variability regions are areas of the image where the pixel values (intensity, color, etc.) are relatively similar or consistent.
- **Example**: In an image of a blue sky, the sky area would be a low-variability region because the pixel values (shades of blue) are similar across that region.
- **Algorithm's Behaviour**: The algorithm preserves the details in these regions because the differences between adjacent pixel values are small. It recognizes that these areas are homogeneous and thus keeps them intact as a single coherent segment.

High-Variability Image Regions

- **Definition**: High-variability regions are areas of the image where the pixel values change significantly and frequently.
- **Example**: In an image of a forest, the area with trees, leaves, and shadows would be a high-variability region because the pixel values (different shades of green, brown, etc.) vary greatly and frequently.
- **Algorithm's Behaviour**: The algorithm tends to ignore the fine details in these regions to avoid over-segmentation. It recognizes that these areas are heterogeneous and instead of creating many small segments, it merges them into larger segments to simplify the representation of the image.

Advantages and Disadvantages of Graph-based Segmentation

Factor	Advantages	Disadvantages
Flexibility	Can handle multiple image types and applications. Can segment images based on color, texture, or brightness.	Requires tuning several parameters, such as edge weight and graph partitioning criteria.
Accuracy	May produce accurate and precise results by considering local and global information in an image. Can capture fine details and contours of objects, resulting in more accurate segmentations.	May produce too many segments or over-segmentation if the graph is incorrectly partitioned, resulting in segments that are too small or too numerous for practical use.
Efficiency	Algorithms are fast and efficient. Can handle large images with thousands of pixels in real-time, making them suitable for real-time applications.	Complex technique that requires significant computational resources. Involves constructing a graph, defining edge weights, and partitioning the graph into segments, which may make it hard to implement and optimize.

Explaining the Program Functionality (What is going on in the code):

The program's functionality is described below, in accordance with the theoretical steps for Graph-Based Image Segmentation.
Also, I have provided an example for each step showing how it works
(Program File: CV_ProjectTask2_NikhilKumar.pde)

Importing Required Libraries –
import java.util.*;

- **Purpose**: This imports the Java utility package, which includes the necessary data structures like Array List, HashSet, and HashMap that will be used later in the program.

Global Variables
PImage img; and int threshold;

- **PImage img**: This variable will store the image that we want to segment.
- **int threshold**: This defines the threshold value for the segmentation algorithm. It determines the maximum allowed difference in intensity between pixels for them to be considered part of the same segment.

Setup Function - void setup() :

This is a special method in Processing that runs once at the beginning of the program. It is used for initial setup tasks.

Loading the Image
img = loadImage("NumberPlate.jpg");

Purpose: Load the image named " NumberPlateB.jpg" into the img variable.
Function: load Image is a built-in Processing function that reads the image file and creates a PImage object.

Setting Canvas Size
size(1300, 1400);

Purpose: Sets the size of the canvas to 1300 pixels in width and 1400 pixels in height.
Function: size is a built-in Processing function that specifies the dimensions of the display window.

Displaying the Original Image
image(img, 0, 0);

Purpose: Draws the loaded image on the canvas at the coordinates (0, 0), which is the top-left corner.
Function: image is a built-in Processing function that displays images on the canvas.

Loading Pixel Data
img.loadPixels();

Purpose: Loads the pixel data of the image into the pixels array, allowing for direct manipulation of individual pixels.

Function: loadPixels is a method of the PImage class that prepares the image's pixel data for manipulation.

Performing Segmentation

int[][] segments = graphBasedSegmentation(img, threshold);

Purpose: Calls the graphBasedSegmentation function, passing the image and threshold value. This function returns a 2D array representing the segments in the image.

Function: graphBasedSegmentation is a custom function that applies the graph-based segmentation algorithm to the image.

Printing Number of Segments

println(" Number of segments: " + segments.length);

Purpose: Outputs the number of segments found by the segmentation algorithm to the console.

Function: println is a built-in Processing function that prints messages to the console.

Updating Pixel Data

img.updatePixels();

Purpose: Updates the pixels array of the image to reflect any changes made during segmentation.

Function: updatePixels is a method of the PImage class that updates the image with any modifications made to its pixel data.

Displaying Segmented Image

image(img, 0, img.height + 30);

Purpose: Draw the segmented image on the canvas below the original image. The y-coordinate is set to img. height + 30 to position the segmented image 30 pixels below the original.

Function: This second call to image displays the modified image, showing the segmentation results.

Segmentation Function:

int[][] graphBasedSegmentation(PImage image, int threshold)

- **Purpose**: This method performs graph-based segmentation on the input image using the specified threshold.
- **Parameters**:
  - PImage image: The image to be segmented, is represented as a PImage object.
  - int threshold: The threshold value used to determine whether pixels should be merged into the same segment.

Extracting Image Dimensions

int w = image.width: Retrieves the width of the image and assigns it to the variable w.
int h = image.height: Retrieves the height of the image and assigns it to the variable h.

Calculating the Total Number of Pixels
int numPixels = w * h;
The total number of pixels in the image is calculated by multiplying its width (w) and height (h). Knowing the total number of pixels is crucial for initializing data structures containing information about each pixel and for iterating over all pixels in the image.

Steps Performing Image Segmentation Algorithm with examples:

Step 1: Initialize each pixel as its own segment

This block of code initializes an array where each pixel in the image is initially assigned to its own unique segment. This is the starting point for the segmentation algorithm, where each pixel is considered an individual segment.

int[] segments = new int[numPixels];

- **Purpose**: Declares an array named segments with a length equal to the total number of pixels in the image (numPixels).
- **Function**: This array will be used to keep track of which segment each pixel belongs to. Initially, every pixel is its own segment

Initializing Segments

```
for (int i = 0; i < numPixels; i++)
{
   segments[i] = i;
}
```

- **Purpose**: Initializes each element of the segments array.
- **Function**: The loop iterates through each pixel in the image:
  - for (int i = 0; i < numPixels; i++): The loop runs from 0 to numPixels - 1, covering all pixels in the image.
  - segments[i] = i: Assigns each pixel to a unique segment by setting the value of segments[i] to i. This means that initially, each pixel is considered its own segment.

Example of creating segments

If `numPixels` is `5`, the array `segments` will be initialized and populated as follows:

- Before the loop: `segments = [0, 0, 0, 0, 0]`
- After the first iteration (i=0): `segments = [0, 0, 0, 0, 0]`
- After the second iteration (i=1): `segments = [0, 1, 0, 0, 0]`
- After the third iteration (i=2): `segments = [0, 1, 2, 0, 0]`
- After the fourth iteration (i=3): `segments = [0, 1, 2, 3, 0]`
- After the fifth iteration (i=4): `segments = [0, 1, 2, 3, 4]`

Creating a List:

```
List<Edge> edges_connected = new ArrayList<>();
```

- **Purpose**: Declares a list named edges_connected to store the edges between pixels.
- **Function**: This list will hold Edge objects, each representing a connection between two adjacent pixels with a weight based on their intensity difference.

Nested Loop for Iterating Over Pixels

```
for (int y = 0; y < h; y++)
{
   for (int x = 0; x < w; x++)
   {
      int current = x + y * w;
   }
}
```

- **Purpose**: The outer loop iterates over each row (y) of the image, and the inner loop iterates over each column (x).
- **Function**: This nested loop ensures that every pixel in the image is considered for edge creation.

Creating Horizontal Edges

```
if (x < w - 1)
{
   int next = (x + 1) + y * w;
   edges_connected.add(new Edge(current, next, weight(image.pixels[current],
   image.pixels[next])));
}
```

- **Condition**: if (x < w - 1) ensures that the current pixel is not on the right edge of the image.
- **Next Pixel**: Calculates the index of the next pixel to the right.
- **Edge Creation**: Creates an edge between the current pixel and the next pixel to the right, and adds it to the edges_connected list.
- **Weight Calculation**: The weight of the edge is calculated as the difference in intensity between the two pixels.

Creating Vertical Edges

```
if (y < h - 1)
{
   int next = x + (y + 1) * w;
   edges_connected.add(new Edge(current, next, weight(image.pixels[current],
   image.pixels[next])));
}
```

- **Condition**: if (y < h - 1) ensures that the current pixel is not on the bottom edge of the image.
- **Next Pixel**: Calculates the index of the next pixel below.
- **Edge Creation**: Creates an edge between the current pixel and the next pixel below, and adds it to the edges_connected list.
- **Weight Calculation**: The weight of the edge is calculated as the difference in intensity between the two pixels.

Creating Diagonal Edges (Top-Left to Bottom-Right)

```
if (x < w - 1 && y < h - 1)
{
   int next = (x + 1) + (y + 1) * w;
   edges_connected.add(new Edge(current, next, weight(image.pixels[current],
   image.pixels[next])));
}
```

- **Condition**: if (x < w - 1 && y < h - 1) ensures that the current pixel is not on the right or bottom edges of the image.
- **Next Pixel**: Calculates the index of the next pixel diagonally (top-left to bottom-right).
- **Edge Creation**: Creates an edge between the current pixel and the next diagonal pixel, and adds it to the edges_connected list.
- **Weight Calculation**: The weight of the edge is calculated as the difference in intensity between the two pixels.

Step 2: Sort edges in the order of non-decreasing edge weight

Sorting Edges by Weight

```
Collections.sort(edges_connected);
```

- **Purpose**: This line of code sorts the list of edges based on their weights in ascending order.
- **Function**: Sorting the edges allows for efficient processing during the segmentation algorithm, as edges with lower weights (representing smaller intensity differences) are processed first.

Detailed Explanation

1. **Using Collections Utility Class**
   - ❖ Collections is a utility class in Java that provides various methods to operate on collections, such as sorting.
   - ❖ sort is a static method of the Collections class used to sort a list of elements.

2. **Sorting Edges**
   - ❖ sort(edges_connected): This method sorts the edges_connected list in ascending order.
   - ❖ The sorting is performed based on the natural ordering of the Edge objects. Since the Edge class implements the Comparable interface and defines a **compareTo method**, Java's default sorting algorithm uses this method to compare edges and sort them accordingly.

3. **Sorting Mechanism**:
   - ❖ The `Collections.sort()` method is employed to sort the list of edges (`edges_connected`) based on their weights.
   - ❖ Internally, Java's sorting algorithm uses the `compareTo()` method implemented in the `Edge` class to compare the weights of different edges and arrange them in ascending order.

4. **Invoking the `compareTo` Method**: When `Collections.sort()` is called, it internally invokes the **`compareTo`** method of the `Edge` class to compare `Edge` objects. In the `compareTo` method, `this.weight` represents the `weight` of the current `Edge` object (the one being compared), and `other.weight` represents the `weight` of the `Edge` object passed as an argument to the `compareTo` method.

   The **`compareTo`** method returns a negative integer, zero, or a positive integer depending on whether `this.weight` is less than, equal to, or greater than `other.weight`, respectively.

So, to summarize:

- When a new `Edge` object is created, its `compareTo` method is not invoked.
- The `compareTo` method is invoked when comparing `Edge` objects for sorting purposes, typically using methods like `Collections.sort()`.
- In the `compareTo` method, `this.weight` represents the `weight` of the current `Edge` object, and `other.weight` represents the `weight` of the `Edge` object being compared

Step 3: Merging the Segments and Colouring

1. Iterating Over Edges

for (Edge edge : edges_connected)

- **Purpose**: Iterate over each edge in the edges_connected list. Each edge represents a connection between two pixels with a weight indicating the intensity difference.

2. Checking the Weight Against the Threshold

if (edge.weight <= threshold)

- **Purpose**: Only process the edge if its weight is less than or equal to the threshold. This threshold determines how similar the pixel intensities should be for them to be considered part of the same segment.

3. Getting the Segment Identifiers

int set1 = segments[edge.u];
int set2 = segments[edge.v];

Purpose: Retrieve the current segment identifiers for the two pixels connected by the edge. edge.u and edge.v are the indices of the two pixels, and segments is an array where each index represents a pixel and the value represents the segment it belongs to.

4. **Checking if the Pixels are in Different Segments**

if (set1 != set2)
Purpose: Determine if the two pixels are in different segments. If they are, they need to be merged into one segment.

5. **Merging Segments**

```
for (int i = 0; i < numPixels; i++)

{
    if (segments[i] == set2)
       {
           segments[i] = set1;
        }
 }
```

- **Purpose**: Merge the segments by changing all pixels in set2 to be part of set1.
  - **Loop**: Iterate over all pixels.
  - **Condition**: Check if the current pixel belongs to set2.
  - **Action**: If it does, change its segment identifier to set1. This effectively merges set2 into set1.

Example of Merging

Assume we have a simple image with 4 pixels and initially, each pixel is its own segment:
`segments = [0, 1, 2, 3]`

And the edges are:
`edges_connected = [Edge(0, 1, 5), Edge(1, 2, 10), Edge(2, 3, 20)]`

With a threshold of `15`:

- Edge (0, 1) with weight 5 is processed because `5 <= 15`:
  `set1 = segments[0] = 0`
  `set2 = segments[1] = 1`

Since `set1 != set2`, merge segments:
`segments = [0, 0, 2, 3]`

- Edge (1, 2) with weight 10 is processed because `10 <= 15`:
  `set1 = segments[1] = 0`
  `set2 = segments[2] = 2`

Since `set1 != set2`, merge segments:
`segments = [0, 0, 0, 3]`

Edge (2, 3) with weight 20 is not processed because `20 > 15`.

Final segments: **`segments = [0, 0, 0, 3]`**

1. Collect Unique Segments

```
Set<Integer> uniqueSegments = new HashSet<>();

for (int segment : segments)
{
 uniqueSegments.add(segment);
}
```

Purpose: To collect unique segment identifiers from the segments array.

- **HashSet Initialization**: unique segments is a HashSet that will store unique segment identifiers. The HashSet does not allow duplicate elements, which helps in collecting only unique segment values.
- **Loop**: Iterate over each segment in the segments array.
- **Add to HashSet**: Each segment identifier is added to the HashSet. Since HashSet only stores unique values, duplicates are automatically ignored.

2. Assign Color RGB Values to Segments

```
Map<Integer, Integer> segmentColors = new HashMap<>();

for (int segment : uniqueSegments)

{
 segmentColors.put(segment, color(random(255), random(255), random(255)));
}
```

Purpose: To assign random RGB colors to each unique segment.

- **HashMap Initialization**: segmentColors is a HashMap where the key is the unique segment identifier and the value is the colour assigned to that segment.
- **Loop**: Iterate over each unique segment in the uniqueSegments set.
- **Assign Random Color**: For each segment, generate a random color using the color(random(255), random(255), random(255)) function and put it in the HashMap.

3. Assign Segment Colors to All Pixels

```
for (int i = 0; i < numPixels; i++)
{
img.pixels[i] = segmentColors.get(segments[i]);
}
```

Purpose: To update the image pixels with the colors assigned to their respective segments.

- Loop: Iterate over each pixel in the image.
- Assign Color: For each pixel, retrieve the color from segmentColors using the segment identifier (segments[i]) and assign it to the pixel in the img.pixels array.

Practical Example of HashSet, HashMap, and Updating pixels with colours:
Suppose you have the following segment identifiers for pixels:

segments = {0, 0, 1, 1, 2, 2, 0, 0, 1, 1, 2, 2};

- **Unique Segments**:
  - The HashSet will contain: {0, 1, 2}.
- **Assign Colors**:
  - Assume the HashMap assigns colors as follows:
    - Segment 0 -> Color #FF0000 (Red)
    - Segment 1 -> Color #00FF00 (Green)
    - Segment 2 -> Color #0000FF (Blue)
- **Update Pixels**:
  - For each pixel, update its color based on its segment:
    - Pixels in segment 0 will be colored Red.
    - Pixels in segment 1 will be colored Green.
    - Pixels in segment 2 will be colored Blue.

The process ensures that all pixels belonging to the same segment (identified by a unique segment identifier) are assigned the same color. Duplicates are handled implicitly by grouping pixels into unique segments first and then coloring them based on these unique groups

4. Convert Segments to Array

int[][] segmentsArray = new int[uniqueSegments.size()][];

- **Purpose**: To create a two-dimensional array to hold the pixel indices for each unique segment.
  Array Initialization: segmentsArray is initialized with a size equal to the number of unique segments. Each element of segmentsArray will later hold an array of pixel indices that belong to the same segment.

Loop Through Unique Segments
int i = 0;
for (int segment : uniqueSegments)

Purpose: To iterate through each unique segment and collect the indices of pixels that belong to that segment.
Index Initialization: i is used to keep track of the current position in segmentsArray.

Collect Pixel Indices for Each Segment

```
List<Integer> pixelList = new ArrayList<>();
for (int j = 0; j < numPixels; j++)
{
    if (segments[j] == segment)
  {
      pixelList.add(j);
   }
}
```

Purpose: To collect the indices of pixels belonging to the current segment.

- **Pixel List Initialization**: pixelList is created to store the indices of pixels that belong to the current segment.
- **Inner Loop**: Iterate through each pixel.
- **Condition Check**: If the pixel's segment matches the current segment, add its index to pixelList.

Weight Function: This function determines the weight, or difference, between two pixels based on their brightness.

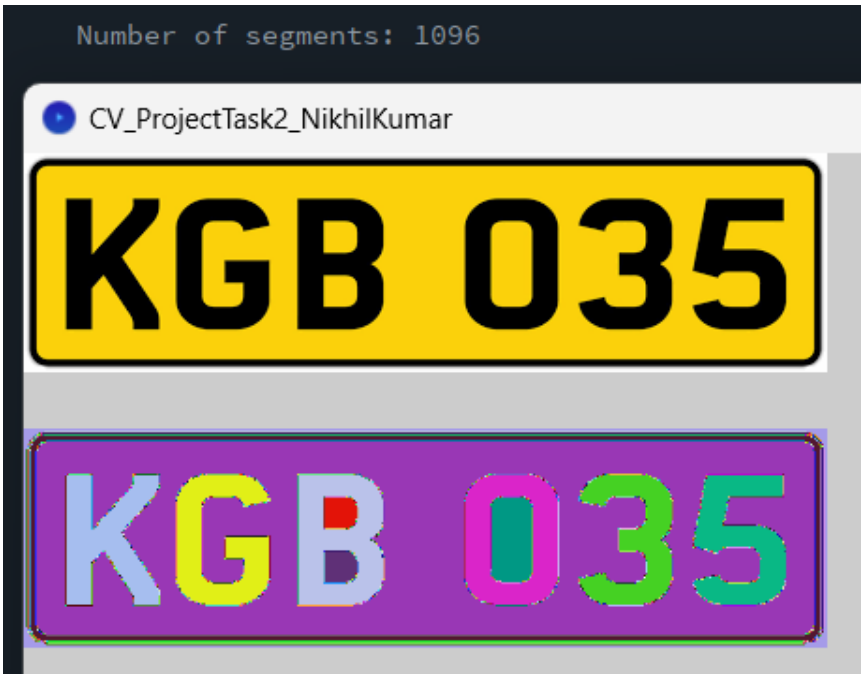
- Parameters:
  - color1: The color value of the first pixel.
  - color2: The color value of the second pixel.
- Steps:
  - brightness(color1): This function internally converts the color value of the first pixel to its corresponding brightness (intensity), effectively converting it to a gray pixel.
  - brightness(color2): Similarly, this function converts the color value of the second pixel to its brightness, also resulting in a gray pixel.
  - abs(intensity1 - intensity2): Here, the absolute difference between the brightness values of the two gray pixels is calculated, representing the weight of the edge connecting these two pixels in the graph.
- Return Value: The absolute difference in brightness between the two pixels, which serves as the weight of the edge.

Edge Class: This class represents an edge in a graph, connecting two vertices (or pixels) with a specified weight.

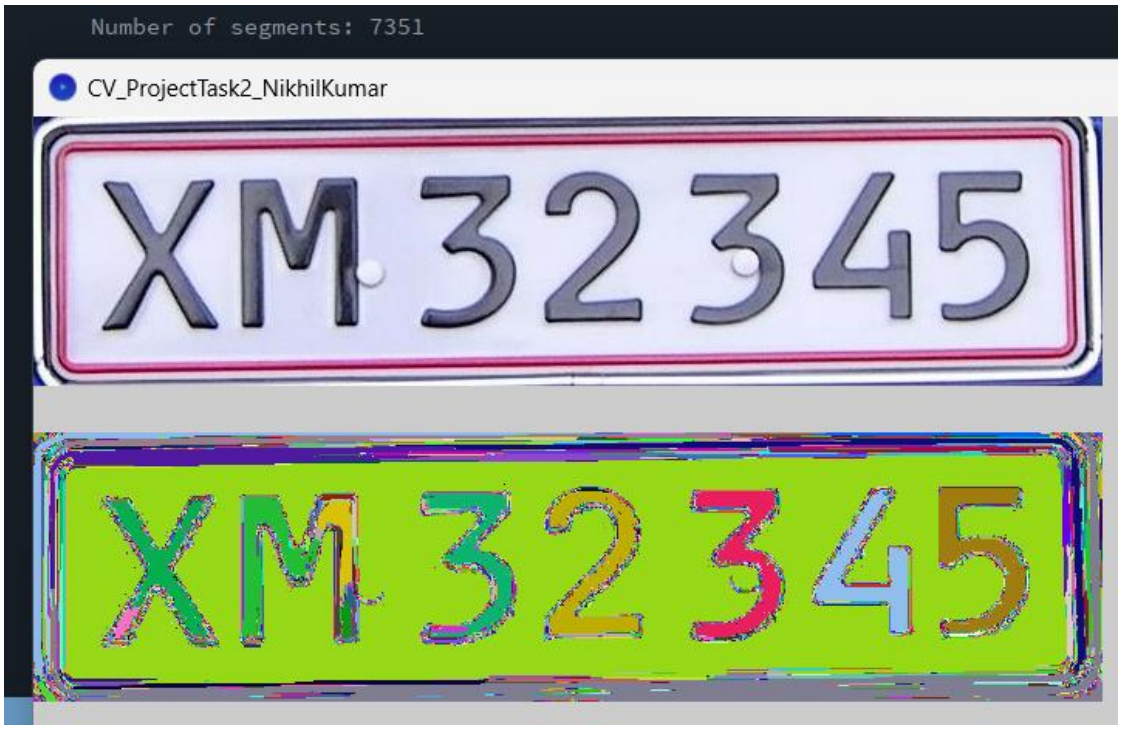
- **Instance Variables**:
  - int u: One vertex of the edge.
  - int v: The other vertex of the edge.
  - float weight: The weight of the edge, typically representing the similarity or difference between the two vertices.
- **Constructor**:
  - Edge(int u, int v, float weight): Initializes the edge with two vertices and their connecting weight.
    - this.u = u: Assigns the parameter u to the instance variable u.
    - this.v = v: Assigns the parameter v to the instance variable v.
    - this.weight = weight: Assigns the parameter weight to the instance variable weight.
- **compareTo Method**:
  - Implements the Comparable interface, allowing edges to be compared based on their weight.
  - return Float.compare(this.weight, other.weight): Compares the weight of the current edge with another edge, returning a value based on the comparison:
    - Negative if the current edge's weight is less.
    - Zero if the weights are equal.
    - Positive if the current edge's weight is greater.

Output Results

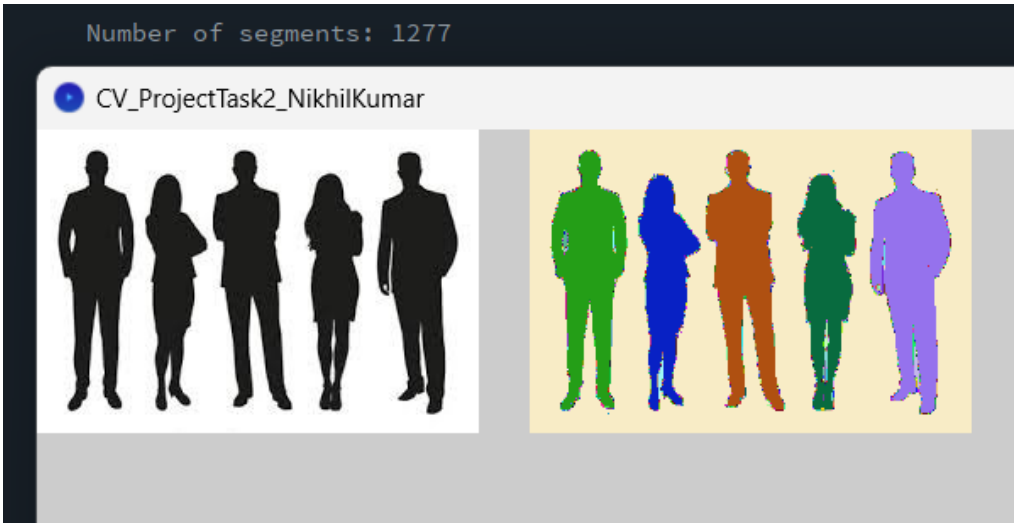
1. Threshold Value = 9



2. Threshold Value = 7



3. Threshold Value = 20



Conclusion

Graph-based image segmentation, as demonstrated in the provided code, is an effective technique for dividing an image into distinct segments based on pixel intensity differences. The process involves several key steps: initializing each pixel as its own segment, creating edges between adjacent pixels, sorting these edges by weight, and merging segments based on a specified threshold. This method excels in preserving details in low-variability regions while simplifying high-variability areas, making it suitable for various image types.

The conversion to grayscale ensures a consistent approach to segmentation, as intensity differences become easier to compute and compare. By assigning unique colors to each segment, the segmented image clearly distinguishes between different regions, enhancing visual interpretation. This approach is computationally efficient and adaptable, capable of processing both grayscale and color images with slight modifications.

Additional Information
Observation of the Difference in Number of Segments for Color and Grayscale Images:

There will be a difference in the number of segments for Color and Grayscale images because-

- In **Grayscale Image** the pixel values range from **0 to 255**, representing intensity.
  Meaning: Each pixel value represents the intensity of light.

    - **0** = Black (no light intensity).
    - **255** = White (full light intensity).
    - **128** = Mid-grey (half light intensity).

In a grayscale image, The segmentation algorithm processes these single-channel intensity values, making the computation of weights (differences in intensity) straightforward. Because there is only one channel to consider

- In **Colour Image**: Each pixel has three values (R, G, B), each ranging from **0 to 255**, representing the intensities of red, green, and blue channels.
  Meaning: Each value corresponds to the intensity of the red, green, and blue components of that pixel.

    - **(0, 0, 0)** = Black (no intensity in any channel).
    - **(255, 255, 255)** = White (full intensity in all channels).
    - **(255, 0, 0)** = Red (full intensity in the red channel, no intensity in the green and blue channels).
    - **(0, 255, 0)** = Green (full intensity in the green channel, no intensity in the red and blue channels).
    - **(0, 0, 255)** = Blue (full intensity in the blue channel, no intensity in the red and green channels).

In a color image, each pixel is represented by three values (R, G, B), each ranging from 0 to 255. The segmentation algorithm processes these three-channel values, which adds complexity to the computation of weights. The weight function must consider the differences in all three color channels, which can lead to different edge weights compared to grayscale images

Reference:

1. **Text Reference:**

a) https://www.analyticsvidhya.com/blog/2021/05/image-segmentation-with-felzenszwalbs-algorithm/
b) https://medium.com/@jonas.cleveland/best-image-segmentation-models-a8620935b5b8#:~:text=Threshold%2Dbased%20segmentation%2C%20graph%2D,the%20best%20image%20segmentation%20models.
c) https://www.linkedin.com/pulse/image-segmentation-10-concepts-5-use-cases-hands-on-guide-updated/
d) https://www.oreilly.com/library/view/hands-on-image-processing/9781789343731/a30eef15-3f00-48b5-982a-e2bee047ba81.xhtml#:~:text=Felzenszwalb's%20algorithm%20takes%20a%20graph,%2C%20the%20difference%20in%20intensity.
e) Lecture Notes - Gnjatovic.image.processing.pdf
f) https://www.baeldung.com/cs/graph-based-segmentation#how-does-graph-based-segmentation-work
g) https://cs.brown.edu/people/pfelzens/papers/seg-ijcv.pdf

2. **Source Code Reference :**

a) Code Snippets
b) Lecture Notes

3. **Image Reference:**

References of Image used in the Code:

- https://t3.ftcdn.net/jpg/02/30/40/42/360_F_230404295_XkJmajXRI1eBj1DY0qZRD89mbzoX2Q0P.jpg
- https://upload.wikimedia.org/wikipedia/commons/thumb/4/48/Danish_number_plate.jpg/640px-Danish_number_plate.jpg