Image Segmentation With Using Minimum Spanning Trees

Burak Ersoz

1 Abstract

In this project, one algorithm for image segmentation is implemented, aiming to partition an image into distinct regions. A graph-based representation of the image is used to do the image segmentation. Despite its greedy decision-making process, the algorithm consistently generates segmentations that preserve its global properties, ensuring coherence and meaningfulness in the resulting regions. Notably, algorithm runtime complexity is nearly linear in the number of graph edges, it is practical for real-world applications.

2 Graph-Based Segmentation

Converting an image into a graph involves representing the image as a network of interconnected nodes and edges. First, each pixel in the image is considered a node in the graph. These nodes represent the basic units of information in the image. Next, edges are defined to connect neighboring pixels in the image grid. The type of edges and their weights depend on the specific criteria chosen for graph construction. In this project to establish connections between edges and assign weights in the graph representing the image, each pixel is linked to its eight neighboring pixels, considering both horizontal, vertical, and diagonal connections. The weight of each edge is determined by computing the difference between the RGB values of the respective pixels. Specifically, the Euclidean distance metric is employed to quantify the dissimilarity in color space, providing a reliable measure of the distance between RGB vectors. There are other ways to connect nodes, it can be done by setting a constant value and connecting each node whose Euclidean distance in (row, column,r,g,b) is smaller than this constant value. In this project, the 8-way connection is chosen because it is a very fast way to find the neighbors of the node. The image-to-graph process effectively captures the spatial relationships and structural information present in the image. Once the graph is constructed, it can be used to do image segmentation. Overall, transforming an image into a graph provides a versatile framework for understanding and processing visual data.

3 Image Matrix Definition

Let I be an $n \times m$ matrix representing the image, where each element I_{ij} corresponds to the pixel value at row i and column j. Each pixel value could represent grayscale intensity or RGB color values depending on the image type.

4 Graph Definition

Let $G = (V, E)$ be the graph representing the image, where:

- V is the set of nodes representing pixels in the image, denoted as (i, j) for $1 \leq i \leq n$ and $1 \leq j \leq m$.
- E is the set of edges connecting neighboring pixels.

5 Edge Definition

Edges between nodes (i, j) and (k, l) exist if and only if:

- $i = k$ and $|j - l| = 1$ (horizontal neighbors),
- $j = l$ and $|i - k| = 1$ (vertical neighbors), or
- $|i - k| = 1$ and $|j - l| = 1$ (diagonal neighbors).

The weight of each edge is assigned as the Euclidean distance between the pixel values:

$$\text{weight}((i, j), (k, l)) = \sqrt{(I_{ij} - I_{kl})^2}$$

6 Gausian Filtering

The Gaussian Smoothing Operator performs a weighted average of surrounding pixels based on the Gaussian distribution. Sigma defines the amount of blurring.[4] Gaussian filter is used to smooth the image before the computing edge weights. It increased the robustness of subsequent segmentation algorithms, ensuring that the segmentation process focuses on meaningful image features. Because while the image is getting bigger it should focus on the bigger parts of the picture than the small pixel information. For gaussian filtering scipy python[3] library is used. While the constant of the function is increasing the blurriness of the image is increasing. High gaussian filter constant values are used more on the bigger image to get better solutions.

7 Pairwise Region Comparison Predicate

This section defines a predicate, D, that determines whether or not a segmentation's two components have a boundary or not. According to the algorithm, if there is a boundary there should be separate components, if there is no boundary the components should be merged and become one component. The predicate will be comparing the dissimilarity between the boundary elements of two components and the inter-component dissimilarity. The internal difference of a component C is the largest weight in the minimum spanning tree of the component.

$$\mathrm{Int}(C) = \max_{e \in \mathrm{MST}(C,E)}(e)$$

The difference between the two components is the minimum weight edge that connects these two components

$$\mathrm{Diff}(C_1, C_2) = \min_{(u,v)\in E, u\in C_1, v\in C_2}(w(u,v))$$

Differences of the components could be calculated differently like comparing the median of the weights or some other functions. But for this algorithm, this difference function is selected and used.

Minimum internal difference is defined as

$$\mathrm{MInt}(C_1, C_2) = \min(\mathrm{Int}(C_1) + \mathrm{t}(C_1), \mathrm{Int}(C_2) + \mathrm{t}(C_2))$$

In this function, the threshold function is defined as

$$\mathrm{t}(C) = k/|C|$$

This threshold function allows the algorithm to consider more local information when the component is small. When the component is bigger consider more the internal difference of the components. While the k value increases at the end the component number of the image decreases, but the pixels in the component are increasing. The pairwise comparison predicate is

$$D(C_1, C_2) = \begin{cases} \text{true}, & \text{if } \mathrm{Dif}(C_1, C_2) > \mathrm{MInt}(C_1, C_2) \\ \text{false}, & \text{otherwise} \end{cases}$$

This function is used to decide whether is there a boundary between two components or not. If there is no boundary then they should be merged and become one component.(Felzenszwalb & Huttenlocher,2004,8,9)

8 Definitions and property

Too fine segmentation means that there is a pair exists such that they have no boundary according to $D(C_1,\ C_2)$. **The refinement of segmentation** is the relationship between two segmentations, denoted as S and T, sharing

the same base set. In this context, T is considered a refinement of S if every component in T is either contained within or equal to some component of S. It is proper refinement if T != S. **Too coarse** segmentation means that there is a segmentation that is a proper refinement of S and that is not too fine.

For any (finite) graph $G = (V, E)$ there exists some segmentation S that is neither too coarse nor too fine. (Felzenszwalb & Huttenlocher,2004,9-10)

9 Algorithm

The algorithm is similar to the Kruskal minimum spanning tree algorithm. However, there is one difference from the Kruskal algorithm. It does not check only if they are in the same component, it also checks if there are boundaries between them too. In that way, it does not connect all nodes into one tree and creates multiple trees according to the function. In this way, it gives a forest as output and this output is the segmentation of the image.

9.1 Algorithm explanation

The input is a graph $G = (V, E)$, with n vertices and m edges. The output is a segmentation of V into components $S = (C_1, ..., C_r)$.

1. Sort E into ord $= (o_1, \ldots, o_m)$ by non-decreasing edge weight.
2. Initial segmentation is S^0 and each vertex is a component by itself.
3. Repeat step 3 for $q = 1, \ldots, m$.
4. Construct S^q with S^{q-1}. $o_q = (u, v)$ check if they are in the same component. If they are in the same component then continue to the next edge. If they are in different components, let's say C_u, C_v, check if $w(o_q) \leq \text{MInt}(C_u, C_v)$ is true. If true, that means this edge is smaller than their internal difference, then they should be in the same component.

According to Felzenszwalb and Huttenlocher(2004) the result is not too fine and not too coarse.(Felzenszwalb & Huttenlocher,2004,12)

10 Running Time Analysis

Let m denote the number of edges in the graph, and n denote the number of vertices.

- $m = O(n)$
  - For one pixel it creates at most 8 edges for it
- Sorting algorithm: $O(m \log m)$

  - Python's default sorting algorithm is used, it can be faster with using counting sort

- Union operation for all edges complexity: $O(m \log m)$
  - There are m edges to process and for each edge algorithm tries to do a union operation. Union operation is done in a union-find data structure and the complexity of union operation is O(log m)
- Keeping track of the maximum edge and sizes of components: $O(1)$ at each union operation

Overall, the complexity of the algorithm is $O(m \log m)$.

References

[1] Jessica Li. (2021). Coil100. Kaggle.https://www.kaggle.com/datasets/jessicali9530/coil100

[2] Felzenszwalb, P. F., & Huttenlocher, D. P. (2004). Efficient graph-based image segmentation. International journal of computer vision, 59, 167-181.

[3] https://scipy.org/

[4] https://www.southampton.ac.uk/msn/book/new_demo/gaussian/

11 Appendix and Code Documentation

11.1 Gaussian Smoothing

```
"""
    Apply Gaussian smoothing to the input image.
"""
def gaussian_smoothing(image, sigma):
    smoothed_image = scipy.ndimage.gaussian_filter(image, sigma)
    return smoothed_image
```

11.2 PNG to Coloured Pixel Matrix

```
"""
    This function converts a PNG image file to a colored pixel matrix,
    applying Gaussian smoothing to the pixel values.
"""
def png_to_coloured_pixel_matrix(file_path,smoothing_constant = 0.8):
    image = Image.open(file_path)
    pixel_matrix = np.array(image)
    if pixel_matrix.shape[2] > 3:
        pixel_matrix = pixel_matrix[:, :, :3]
```

```
    pixel_matrix = gaussian_smoothing(pixel_matrix, smoothing_constant)

    print("Coloured", file_path, "'s shape:", pixel_matrix.shape)
    return pixel_matrix
```

11.3 PNG to RGB Matrices

```
"""
    This function converts a PNG image file to separate red, green, and blue matrices,
    applying Gaussian smoothing to each channel separately.
"""
def png_to_rgb_matrices(file_path,smoothing_constant = 0.8):
    image = Image.open(file_path)
    pixel_matrix = np.array(image)

    red_channel = pixel_matrix[:, :, 0]
    green_channel = pixel_matrix[:, :, 1]
    blue_channel = pixel_matrix[:, :, 2]

    red = gaussian_smoothing(red_channel, smoothing_constant)
    green = gaussian_smoothing(green_channel, smoothing_constant)
    blue = gaussian_smoothing(blue_channel, smoothing_constant)

    return red,green,blue
```

11.4 Pixel Matrix to Image

```
"""
    Converts a pixel matrix to an image and saves it to a file.
"""
def pixel_matrix_to_image(pixel_matrix, output_file):
    pixel_matrix = pixel_matrix.astype(np.uint8)
    image = Image.fromarray(pixel_matrix)
    image.save(output_file)
```

11.5 Find Neighbors 8-Way

```
"""
    Finds neighboring pixels of a given pixel in a 2D grid.
"""
def find_neighbors_8_way(row, col, rows, cols):
    neighbors = []
    for i in range(max(0, row - 1), min(rows, row + 2)):
        for j in range(max(0, col - 1), min(cols, col + 2)):
            if i != row or j != col:
                neighbors.append((i, j))
```

```
    return neighbors
```

11.6 Coloured Pixel Matrix to Edge List

```
"""
    Converts a pixel matrix into a graph represented as an edge list.
"""
def coloured_pixel_matrix_to_edge_list(pixel_matrix):
    def vector_distance(point1, point2):
        """
            Calculate the Euclidean distance between two points in three-dimensional space.
        """
        [x1, y1, z1] = point1.astype(np.int64)
        [x2, y2, z2] = point2.astype(np.int64)
        p = point1
        t = point2
        dx = x2 - x1
        dy = y2 - y1
        dz = z2 - z1


        squared_distance = dx**2 + dy**2 + dz**2

        distance = math.sqrt(squared_distance)

        return distance
    rows, cols, _ = pixel_matrix.shape
    edge_list = []
    pixel_matrix = pixel_matrix.astype(np.int64)
    # Iterate over each pixel
    for row in range(rows):
        for col in range(cols):
            pixel_value = pixel_matrix[row, col]

            neighbors = find_neighbors_8_way(row, col, rows, cols)

            for neighbor_row, neighbor_col in neighbors:
                neighbor_pixel_value = pixel_matrix[neighbor_row, neighbor_col]

                edge_weight = vector_distance(pixel_value,neighbor_pixel_value)

                edge_list.append((edge_weight,(row, col), (neighbor_row, neighbor_col)))

    return edge_list
```

11.7 Image Segmentation

```
'''
This class provides functionality for segmenting an image into regions based on a graph-base
'''
class ImageSegmentation:

    '''
    Initializes the ImageSegmentation object.
    '''
    def __init__(self, nodes, k):
        self.parent = {}
        self.rank = {}
        self.size = {}
        self.max_edge = {}
        self.image_size = (0,0)
        self.k = k

        for node, value in nodes:
            self.parent[node] = node
            self.rank[node] = 0
            self.size[node] = 1
            self.max_edge[node] = value
            self.image_size = max(self.image_size,node)

   '''
    Computes the threshold function for a given component size.
    '''
    def t_f(self, size_c):
        return self.k / size_c
    '''
    Finds the root of the component containing node x using path compression.
    '''
    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
    '''
    Computes the minimum internal difference between two components.
    '''
    def Mint(self, root_x, root_y):
        size_x = self.size[root_x]
        size_y = self.size[root_y]
        return min(self.t_f(size_x) + self.max_edge[root_x],
                   self.t_f(size_y) + self.max_edge[root_y])
    '''
```

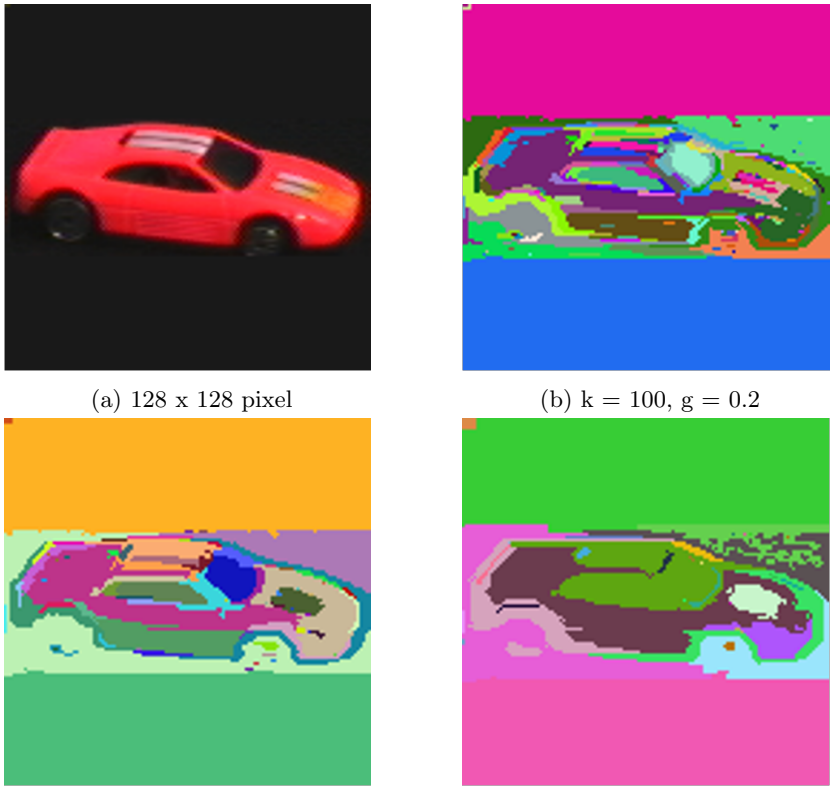
```
  Performs the union operation to merge two components if the edge satisfies the minimum
'''
def union(self,edge_value, x, y):
    root_x = self.find(x)
    root_y = self.find(y)

    if root_x == root_y:
        return

    if self.Mint(root_x, root_y) < edge_value:
        return

    if self.rank[root_x] < self.rank[root_y]:
        self.parent[root_x] = root_y
        self.size[root_y] += self.size[root_x]
        self.max_edge[root_y] = edge_value
    elif self.rank[root_x] > self.rank[root_y]:
        self.parent[root_y] = root_x
        self.size[root_x] += self.size[root_y]
        self.max_edge[root_x] = edge_value
    else:
        self.parent[root_y] = root_x
        self.rank[root_x] += 1
        self.size[root_x] += self.size[root_y]
        self.max_edge[root_x] = edge_value
'''
  Retrieves the groups of nodes after segmentation.
'''
def get_groups(self):
    groups = {}
    for node in self.parent:
        root = self.find(node)
        if root not in groups:
            groups[root] = []
        groups[root].append(node)
    return list(groups.values())


'''
      Generates an image matrix with each pixel colored according to its component.
'''
def get_image_matrix_with_components(self):
    max_x = self.image_size[1] + 1
    max_y = self.image_size[0] + 1

    root_to_color = {}
```

```
        unique_roots = set(self.parent.values())
        unique_colors = set()
        for root in unique_roots:
            r = random.randint(0, 255)
            g = random.randint(0, 255)
            b = random.randint(0, 255)
            root_to_color[root] = (r, g, b)

        new_matrix = np.zeros((max_y, max_x, 3), dtype=int)

        for node, root in self.parent.items():
            new_matrix[node[0], node[1]] = root_to_color[root]
        return new_matrix

    '''
      Runs the segmentation algorithm on the sorted list of edges.
    '''
    def run_segmentation(self,sorted_edges):
        for edge in sorted_edges:
            self.union(edge[0],edge[1],edge[2])
```

11.8 main function

```
import image_service as ser
import image_segmentation as un
import numpy as np
import os

folder_path = "images/"
file_paths = file_names = os.listdir(folder_path)

#file_paths = ["street.png"]
gaus_values = [i / 10.0 for i in range(0, 10,2)]
k_values = [i*100 for i in range(1, 10,2)] + [1000,1500,3000,10000]

k_values = [500]
gaus_values = [0.9]

#rgb segmentation

for file_name in file_paths:
    for k in k_values:
        for gaus in gaus_values:
            file_path = folder_path+file_name
            pixel_matrix = ser.png_to_coloured_pixel_matrix(file_path,gaus)
            #print(pixel_matrix)
```

```
rows, cols,_ = pixel_matrix.shape

edge_list = ser.coloured_pixel_matrix_to_edge_list(pixel_matrix)
edge_list.sort()

segmentation = un.ImageSegmentation([((x,y),0) for x in range(rows) for y in ra
                   ,k=k)
segmentation.run_segmentation(edge_list)
seg_count = len(segmentation.get_groups())
print(f"seg_count = {seg_count}")
segmented_matrix = segmentation.get_image_matrix_with_components()
if not os.path.exists("seg_"+file_name):
    os.makedirs("seg_"+file_name)
output_file_name = "seg_"+file_name+"/k="+str(k)+"_g="+str(gaus)+"_seg="+str(seg
output_file = output_file_name
ser.pixel_matrix_to_image(segmented_matrix, output_file)
```



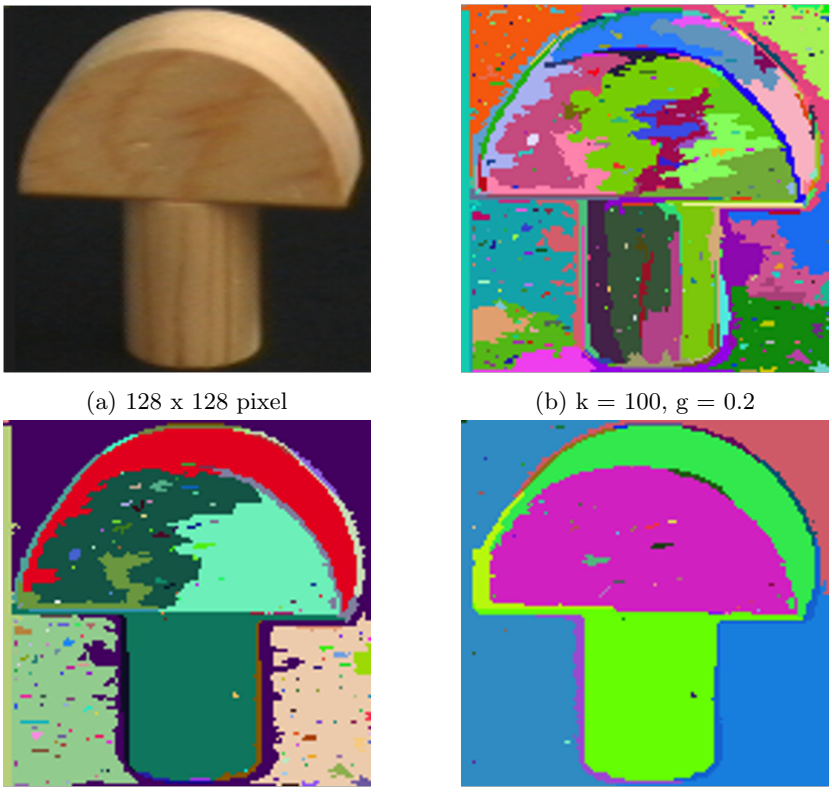
(a) 128 x 128 pixel

(b) k = 100, g = 0.2

(c) k = 300, g = 0.2

(d) k = 300, g = 0.8

Figure 1: Example car image from coil database[1] and segmentation with different k and gaussian filter constants



(a) 128 x 128 pixel

(b) k = 100, g = 0.2

(c) k = 300, g = 0.2

(d) k = 300, g = 0.8

Figure 2: Example wooden mushroom image from coil database[1] and segmentation with different k and gaussian filter constants

(a) 432 x 294 pixel

(b) k=1500 , g = 0.9

(c) k=100000, g=0.9

Figure 3: Example baseball image from article [2] and segmentation with different k and gaussian filter constants