Δ05 Computer Vision 2022-2023

Nikou Christoforos

Assignment 3 – Graph based agglomerative image segmentation

Kotsinas Georgios AM 471

Introduction

Clustering algorithms play a vital role in analyzing and organizing data by identifying similarities between data items. However, comparing all pairs of data items may not always be practical or advantageous, especially when dealing with disparate image pixels. To address this challenge, the concept of representing data items as vertices in a graph arises naturally. A weighted edge is established between data items that exhibit meaningful comparability. Consequently, the process of clustering the data can be approached as the segmentation of the graph into connected components, facilitating effective image segmentation. In this report, we explore the application of graph-based agglomerative method for image segmentation, which leverage the inherent structure and relationships within an image to merge similar regions iteratively, yielding visually coherent and meaningful segments. The report will be structured into three sections. Section 1 will delve into the algorithmic aspects, section 2 will present the experimental results and section 3 will provide a code description.

Algorithm

In their work, Felzenszwalb and Huttenlocher (2004) introduced a graph-based approach for image segmentation, leveraging concepts from graph theory to build a straightforward clusterer. The idea involves representing the image as a weighted graph, where pixels serve as vertices and neighboring pixels are connected by edges. Dissimilarity between pixels is quantified by assigning weights to these edges, with larger weights indicating greater dissimilarity and smaller weights representing similarity.

The dissimilarity weights can be computed using various pixel representations. One common strategy involves calculating the squared difference in intensity between pixels for gray-scale images. Another approach is to represent the color at each pixel as a vector and measure the length of the difference vector. Texture information can also be incorporated by representing pixel texture as a vector of filter outputs and utilizing the length of the difference vector. Alternatively, a weighted combination of these distance measures can be employed. In this particular approach each pixels is represented by an rgb vector, the distance between two rgb vectors converted into an affinity with the help of a generalized Gaussian kernel. The weight of the edge becomes as follows. Where distance, the euclidean distance between two rgb vectors and where σ, the standard deviation.

$$w_{ij} = \exp\left(-\frac{1}{2\sigma^2} \mathrm{dist}(x_i, x_j)^2 \right)$$

To facilitate the merging process, initially each pixel is considered as an individual cluster. Then, clusters are merges iteratively until further merging is unnecessary. To determine the mergeability of two clusters, we require a measure of the distance between them. In this context, a cluster corresponds to a component of the graph, comprising all the vertices (pixels) within the cluster and the edges confined within it. The dissimilarity between two components is quantified by identifying the minimum weight edge that connects them. For ease of notation, let C1 and C2 represent the two components, E denote the edges, and w(v1, v2) denote the weight of the edge connecting vertices v1 and v2. Then:

$$\mathrm{diff}(C_1, C_1) = \min_{v_1 \in C_1, v_2 \in C_2} w(v_1, v_2)$$

Furthermore, it is valuable to assess the coherence of individual clusters to determine when to stop the clustering process. To quantify the coherence of a specific component, so the concept of internal difference is intoduced. The internal difference of a component is defined as the maximum weight among the edges in the minimum spanning tree of that component. Let M(C) = {V_C, E_M} represent the minimum spanning tree of component C, where V_C denotes the vertices of C and E_M denotes the edges of the minimum spanning tree. Then:

$$\mathrm{int}(C) = \max_{e \in M(C)} w(e)$$

The segmentation process begins with a set of clusters, where each cluster corresponds to an individual pixel-segment. Through iterative steps, these clusters are merged to form larger segments. To accomplish this, all edges are sorted based on their non-decreasing weights. Starting from the smallest weight, each edge is evaluated by examining the clusters at both ends of the edge. If the edge connects pixels within the same cluster, no action is taken. However, if the edge connects distinct clusters, a merge operation is considered. The merge occurs when the edge weight is relatively small compared to the internal difference of each cluster, although special consideration is required for small clusters (explained in detail below). This process continues through all the edges, performing the necessary merges. The final segmentation corresponds to the set of clusters once the last edge has been processed.

Start with a set of clusters $\mathcal{C}_i$, one cluster per pixel.
Sort the edges in order of non-decreasing edge weight, so that $w(e_1) \geq w(e_2) \geq \ldots \geq w(e_r)$.

For $i = 1$ to r
    If the edge e_i lies inside a cluster
        do nothing
    Else
        One end is in cluster $\mathcal{C}_l$ and the other is in cluster $\mathcal{C}_m$
        If $diff(\mathcal{C}_l, \mathcal{C}_m) \leq MInt(\mathcal{C}_l, \mathcal{C}_m)$
            Merge $\mathcal{C}_l$ and $\mathcal{C}_m$ to produce a new set of clusters.

Report the remaining set of clusters.

When comparing the edge weight to the internal difference of the clusters, it is important to handle small clusters cautiously. In such cases, the internal distance may be zero (if there is only one vertex) or unreasonably small. To address this issue, Felzenszwalb and Huttenlocher (2004) introduced a function that takes two clusters into account. Mint, as

$$\mathrm{MInt}(C_1, C_2) = \min\left(\mathrm{int}(C_1) + \tau(C_1), \mathrm{int}(C_2) + \tau(C_2)\right)$$

The term, τ(C), introduces a bias to increase the internal difference for such clusters. Specifically, they employ the expression τ(C) = k/|C|, where k is a constant parameter and C the number of nodes the cluster has. By incorporating this term, the internal difference of small clusters is adjusted upwards, allowing for a more appropriate comparison between the edge weight and the internal difference during the merging process.

Experiments

This section presents the experimental results conducted to evaluate the proposed approach, wherein the values of several parameters were systematically varied. The parameters under consideration include the neighborhood hops, which determine the extent to which edges are formed between pixels and their neighbors, encompassing horizontal, vertical, and diagonal directions. Additionally, the standard deviation and the constant parameter k of the function τ(C) were adjusted. For each image, multiple combinations of parameter values were explored until suitable outcomes were achieved. Notably, smaller images were selected for experimentation purposes, considering the relatively time-consuming nature of the algorithm. The images used in this this evaluation were sourced from https://unsplash.com/. Each segment presented in the outcome image with different color, the selection of the color is random each time and has nothing to do with the parameters or the particular segment itself.

- Image size 100x80 pixels. Neighborhood hops away 1, standard deviation 2 and k = 200.


- Image size 140x70 pixels, neighborhood hops 1, standard deviation 3. Pink image-outcome k = 150, green image-outcome k = 300.



- Image size 100x140 pixels, neighborhood hops 1, standard deviation 3 and k = 400.


neighborhood hops 2, standard deviation 5, k = 400 for the first outcome and k = 500 for the second one


Code description

The code implementation comprises four Python files, each serving a specific purpose. The first file, rgb_vectors.py, contains functions responsible for converting an image into a graph representation, adding edges to the graph, calculating edge weights, and sorting edges based on their weights. The second file, diff_mist.py, encompasses functions dedicated to finding the minimum weighted edge connecting two graph components, identifying the largest weighted edge in the minimum spanning tree of a graph component, and calculating Mint(C1, C2) as discussed previously. The third file, clustering.py, integrates the functionalities of the preceding files, creating cluster objects and implementing the algorithm described earlier. It orchestrates the step-by-step execution of the graph-based agglomerative image segmentation process. Lastly, the file visualization.py contains functions

that facilitate the visualization of intermediate and final outcomes at different stages of the segmentation process.

References

[1] P. Felzenswalb and D. Huttenlocher. Efficient graph-based image segmentation,

International Journal of Computer Vision, 59(2), 167–181, 2004.

[2] D. Forsyth and J. Ponce (second edition, pages 309-311).

[3] Course slides, segmentation by clustering.