

# Union-Find and Felzenszwalb-Huttenlocher Segmentation Algorithm

## 1. Introduction

<https://yuminlee2.medium.com/union-find-algorithm-ffa9cd7d2dba>

The **Union-Find** (also called *Disjoint Set Union* or DSU) is a data structure that efficiently tracks elements divided into multiple disjoint sets. It provides fast operations to find which set an element belongs to, and to merge two sets. This structure is fundamental in algorithms that deal with grouping or connectivity problems, such as Kruskal's Minimum Spanning Tree, dynamic connectivity, and image segmentation (like Felzenszwalb-Huttenlocher).

## 2. Union-Find Data Structure

### Core Operations

1. **Find(x)**  
Determines which group or set the element  $x$  belongs to.  
Returns the *representative* (root) of the set.
2. **Union(x, y)**  
Merges the sets that contain elements  $x$  and  $y$ .

It is like a **magnet system** — each group of magnets sticks together, and “union” connects two magnet clusters.

### Optimization Techniques

To make Union-Find very efficient, two optimizations are applied:

- **Path Compression**  
When performing  $\text{find}(x)$ , we recursively point each node directly to its root, flattening the tree.  
→ This drastically reduces future lookup times.
- **Union by Rank / Size**  
When merging, always attach the smaller tree under the larger one.  
→ Keeps the tree shallow, improving performance.

## Time Complexity

The time complexity of Union-Find operations is nearly  $O(1)$  per operation, more precisely  $O(\alpha(N))$ , where  $\alpha$  is the *inverse Ackermann function*.

- $\alpha(N)$  grows so slowly that even for the number of particles in the universe,  $\alpha(N) \leq 4$ .  
Thus, it is effectively constant in practice.

Proof: <https://codeforces.com/blog/entry/98275>

## Applications of Union-Find

1. **Kruskal's Algorithm** (Minimum Spanning Tree)
2. **Grid Percolation**
3. **Network Connectivity**
4. **Least Common Ancestor in Trees**
5. **Image Processing** (e.g., region segmentation)

Their implementation can be found here:

[https://github.com/Nama21yo/design\\_patterns\\_gang/blob/main/dsa\\_notes/union-find.ipynb](https://github.com/Nama21yo/design_patterns_gang/blob/main/dsa_notes/union-find.ipynb)

## 3. Kruskal's Algorithm (Minimum Spanning Tree)

### Problem Definition

Given an undirected, **weighted graph**  $G = (V, E)$ , we want to find a **Minimum Spanning Tree (MST)** — a subset of edges that:

- Connects all vertices together (no isolated node),
- Has exactly  $|V| - 1$  edges,
- And the total edge weight is minimal.

### Algorithm Steps

1. Sort all edges by ascending weight.
2. Start with an empty MST.
3. For each edge  $(u, v)$  in sorted order:
  - If  $u$  and  $v$  belong to different sets (using find):
    - Add the edge to the MST.
    - Merge the sets using  $\text{union}(u, v)$ .

4. Stop when the MST contains  $n - 1$  edges.

Union-Find ensures we never create a cycle and efficiently tracks connected components.

## 4. Image Segmentation

**Image segmentation** is the partition of an image into **disjoint regions** that correspond to meaningful areas — e.g., objects, textures, or boundaries.

To achieve human-like perception, segmentation cannot rely only on local criteria (like pixel intensity); it must consider global or region-based consistency.

## 5. Felzenszwalb-Huttenlocher Segmentation Algorithm

The paper: <https://cs.brown.edu/people/pfelzens/papers/seg-ijcv.pdf>

The **Felzenszwalb-Huttenlocher (FH)** algorithm (2004) is a graph-based image segmentation approach. It represents the image as a graph where:

- Each pixel = a node
- Edges connect neighboring pixels (e.g., 4- or 8-connectivity)
- Edge weights = dissimilarity between pixels (e.g., color difference)

The algorithm then merges pixels into regions based on internal and external differences.

### Algorithm Intuition

The idea is to merge two regions **only if they are similar enough** compared to their internal variation.

- Each pixel starts as its own region (component).
- Components are gradually merged based on edge weights.
- A dynamic threshold controls when merging should stop.

### Mathematical Definition

Let:

- $C_1$  and  $C_2$  be two components (regions)
- $\text{Int}(C) = \text{internal difference}$  of component  $C$   
→ maximum edge weight within  $C$
- $\text{Dif}(C_1, C_2) = \text{difference}$  between components  $C_1$  and  $C_2$   
→ minimum edge weight connecting the two

### Merge Criterion:

Two components  $C_1$  and  $C_2$  are merged if:

$$D(C_1, C_2) = \begin{cases} \text{true} & \text{if } Dif(C_1, C_2) > MInt(C_1, C_2) \\ \text{false} & \text{otherwise} \end{cases} \quad (3)$$

8

where the minimum internal difference,  $MInt$ , is defined as,

$$MInt(C_1, C_2) = \min(Int(C_1) + \tau(C_1), Int(C_2) + \tau(C_2)). \quad (4)$$

The threshold function  $\tau$  controls the degree to which the difference between two components must be greater than their internal differences in order for there to be evidence of a boundary between them ( $D$  to be true). For small components,  $Int(C)$  is not a good estimate of the local characteristics of the data. In the extreme case, when  $|C| = 1$ ,  $Int(C) = 0$ . Therefore, we use a threshold function based on the size of the component,

$$\tau(C) = k/|C| \quad (5)$$

- $k$  is a constant controlling the scale (larger  $k \rightarrow$  larger segments)
- $|C|$  is the size (number of pixels) in component  $C$

### Algorithm Steps

1. **Preprocessing**
  - Apply Gaussian smoothing with  $\sigma$  - **sigma**(to reduce noise).
2. **Graph Construction**
  - Build a graph where nodes = pixels.
  - Edges connect neighboring pixels.
  - Weight of edge  $(u, v)$  = Euclidean RGB distance.
3. **Sorting**
  - Sort all edges by weight (in ascending order).

#### 4. Segmentation (Union-Find)

- Initialize Union-Find for all pixels.
- For each edge (u, v):
  - If they belong to different components:
    - Compute internal differences and thresholds.
    - Merge if the merge condition holds.

#### 5. Post-processing

- Merge any components smaller than min\_size.

#### 6. Labeling

- Assign unique labels to each component to form a segmentation map.

### Parameters

Parameter	Meaning	Effect
<b>k</b>	Scale parameter	Larger k $\rightarrow$ larger segments
<b><math>\sigma</math> (sigma)</b>	Gaussian smoothing	Reduces noise and small texture variation
<b>min_size</b>	Minimum segment size	Ensures tiny fragments are merged

### Complexity

- Graph building:  $O(V)$
- Sorting edges:  $O(E \log E)$
- Union-Find operations:  $\approx O(E)$
- Overall complexity:  **$O(E \log E)$**  ( $E \approx 4V$  for 4-connectivity)


### Visual Intuition

- The algorithm behaves like **region growing** guided by **graph edge weights**.
- k controls how aggressively to merge.
- After merging stops, each connected component becomes a segment.

### Applications

- Object recognition and detection
- Image segmentation for computer vision pipelines
- Superpixel generation
- Medical imaging and texture analysis

## References

- Felzenszwalb, P. F., & Huttenlocher, D. P. (2004).  
[\*Efficient Graph-Based Image Segmentation\*. International Journal of Computer Vision \(IJCV\), 59\(2\)](#)
- Yumin Lee, [\*Union-Find Algorithm\* — Medium, 2021](#).
- Codeforces Blog: [“Inverse Ackermann Function” \(2022\)](#).
-  quarter DIP Efficient Graph Based Image Segmentation