Edge Detection In Images

Akshat Patiyal[2023062], Chirag Sehgal[2023176] and Jasjyot Gulati[2023257]

Indraprastha Institute Of Information And Technology (IIITD)

Abstract. :
Edge detection plays a pivotal role in image processing and computer vision, serving as a foundational step for applications in object detection, segmentation, and scene understanding.
An edge is useful because it marks the boundaries and divides of plane, object or appearance from other places things. For pattern recognition it is also an intermediate step in the digital images. An edge consists of pixels with the intensity variations of gray tones which are different from their neighbouring pixels.
In this paper, traditional approaches, including gradient-based methods such as the Sobel and Canny operators, and Laplacian-based techniques like the Laplacian of Gaussian, are discussed in detail. In the end , we present an original albeit primitive algorithm based on BFS (Breadth First Search) Algorithm from Graph Theory to detect both horizontal and vertical edges in a given image.

Keywords: Edge Detection · Image Processing · Breadth First Search · Laplacian Operator · Gradient Operator.

1 Introduction

In digital image processing, an edge represents a significant change in intensity, colour, texture, or shading within an image, often corresponding to object boundaries or transitions between regions. Identifying and analyzing these edges can reveal crucial information about an image's structure, including depth, orientation, size, and surface properties. The edge detection process is, therefore, instrumental in many image processing applications, such as feature detection, feature extraction, segmentation, and object recognition. By isolating these essential structural elements, edge detection helps simplify image data, filtering unnecessary information and allowing for more efficient analysis.

Accurate edge detection, however, is challenging, especially in noisy images. Noise can obscure subtle changes in pixel intensity, making it difficult to identify true edges. A threshold-based approach is often used to detect edges, whereby pixels are classified as edge points based on the magnitude of intensity change. However, setting an optimal threshold can be difficult, as it must account for noise and variability across different image types and conditions.

Standard edge detection methods often involve convolving the image with a specialized operator that is sensitive to large gradients, returning zero in uniform regions and highlighting areas of significant change. Although these classical approaches—such as the Sobel [7], Prewitt, and Canny operators[1]—are widely used, they can be limited by noise sensitivity and challenges in handling complex textures. This paper explores the evolution of edge detection methods, from traditional techniques to modern adaptive and fuzzy rule-based algorithms, with an emphasis on handling noisy and complex image environments.

I. Gradient Methods

Edges are a significant change in image intensity between pixels and hence can be found wherever there is a spike in the rate of change of the image intensity. In other words, we just need to find the first order derivative of the image intensity also known as the gradient[11].

$$\Delta G = \begin{bmatrix} \frac{\delta G}{\delta x} & \frac{\delta G}{\delta y} \end{bmatrix}$$

There will be an edge at every point where there is a non-zero magnitude of gradient. To get the first derivative of the image intensity, we simply need to get its gradient by taking the partial derivatives in the x and y directions. This allows us to easily get the magnitude and orientation of the edges.

The magnitude is given by:

$$\text{Magnitude} = \sqrt{G_x^2 + G_y^2}$$

Where G_x and G_y are the x and y components of the gradient, respectively.

The direction of the gradient is given by:

$$\theta = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

Where θ is the direction of the gradient and is perpendicular to the direction of the edge [11].

I.1 Finite Distance Approximation

To calculate the gradient itself, we can use the finite difference method to get an approximation. We need at least a 2x2 matrix, like the one given below, to get the finite distance approximation since you need at least 2 pixels in each direction to be able to calculate it[11].

$$\begin{bmatrix} I_{i,j+1} & I_{i+1,j+1} \\ I_{i,j} & I_{i+1,j} \end{bmatrix}$$

$$\frac{\delta G}{\delta x} \approx \frac{1}{2}\left((I_{i+1,j+1} - I_{i,j+1}) + (I_{i+1,j} - I_{i,j})\right)$$

$$\frac{\delta G}{\delta y} \approx \frac{1}{2}\left((I_{i+1,j+1} - I_{i+1,j}) + (I_{i,j+1} - I_{i,j})\right)$$

I.2 Kernel Convolution

Another way to calculate the gradient is by getting its convolution with special kernels for edge detection. A kernel, in this context, is a small matrix that is used in convolution operations. To calculate the convolution, the kernel is placed over a pixel such that the pixel coincides with the center of the kernel. Then, all the pixels are multiplied by the element that they overlap with, and all the products are added together. Finally, the sum is divided by the sum of the coefficients of the kernel(or 1 if the sum of the coefficients is equal to 0) to give the gradient of the pixel[8].

$$g_{ij} = \left| \frac{\sum_{i=0}^{n} \left(\sum_{j=0}^{n} f_{ij} * k_{ij} \right)}{K} \right|$$

This is the operation done on each pixel where:

$$\begin{aligned} k_{ij} &= \text{The element in the kernel at position i,j} \\ f_{ij} &= \text{the pixel in the original image corresponding to } k_{ij} \\ n &= \text{dimension of the kernel} \\ K &= \text{sum of all coefficients of the kernel} \\ &\quad \text{or 1 if the sum of the coefficients is 0} \\ g &= \text{the pixel in the filtered image} \end{aligned}$$

You will notice that when this operation is applied to the pixels on the edge of the image, not all of the elements of the kernel overlap with a pixel in the image. The way to handle this case varies with different implementations.

I.3 Noise

The intensity of pixels can vary slightly even when the pixels are not at an edge. These small variations in intensity are called noise. Noise can add a possibility for error in our edge detection. To deal with this noise, we can slightly blur the image by making an image where the intensity of each pixel is the average intensity of the pixels around it using convolution [3].

I.3.1 Mean blur

This type of blur takes the intensity of all the surrounding pixels and makes the new value of the pixel their average value [3].

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

This is a kernel that can be used for this operation.

I.3.2 Gaussian blur

The Gaussian blur uses weights in its kernel to give more importance to the pixels closer to it while making the blur so that the effect of the drastic change in intensity when an edge is encountered doesn't skew the average[3].

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

This is an example of a type of kernel for this blur.

I.4 Edge threshold

After all the magnitudes of the gradient have been calculated, to get the edges of the image, we first need to decide how to select our edge. There are 2 ways to go about this[11].

I.4.1 Single thresholding

The simple approach is to choose a single threshold and only count the values of the magnitude. This method is susceptible to noise and can falsely classify bits of noise as edges.

I.4.2 Hysteresis thresholding

In this approach, we choose 2 thresholds, T0 and T1, such that T0<T1. Any value below T0 is not an edge, and any value above T1 is an edge. The values between T0 and T1 are only an if a neighbouring[11]. Having a range where it is possible to be an edge deals with the possibility of having a false positive.

2 Prewitt Operator

The Prewitt operator uses 2 kernels, one for the x direction and one for the y direction. After calculating the directional derivatives in the x and y directions, a final output is created by making an image with the magnitude of the gradient. G_x is computed using the following kernel:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

G_y is computed using this kernel:

$$G_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Each kernel is made up of 2 filters. One part is the filter required to calculate the derivative and the other is the filter for the mean blur.

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

This type of kernel, which is made up of other filters is called a separable filter. This allows the operator to blur and calculate the derivative in the same step [1].

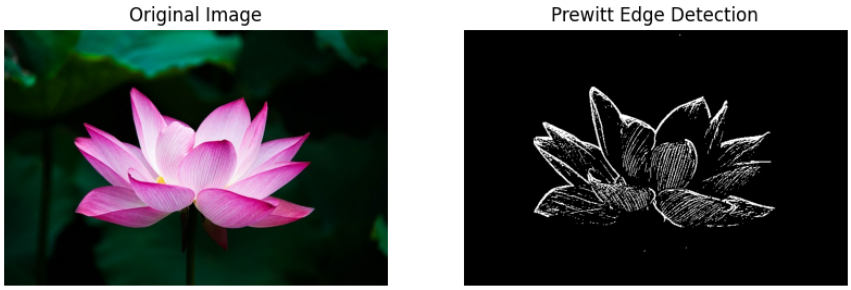

Fig. 1: Example of Prewitt edge detection [12]

3 Sobel Operator

The Sobel operator is similar to the Prewitt operator and it also uses 2 kernels to calculate the derivative. G_x is given by:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

G_y is given by:

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

However, the difference between the 2 operators is that the Sobel operator uses the Gaussian blur to average the pixels. This makes it less sensitive to noise in comparison to the Prewitt operator [1].

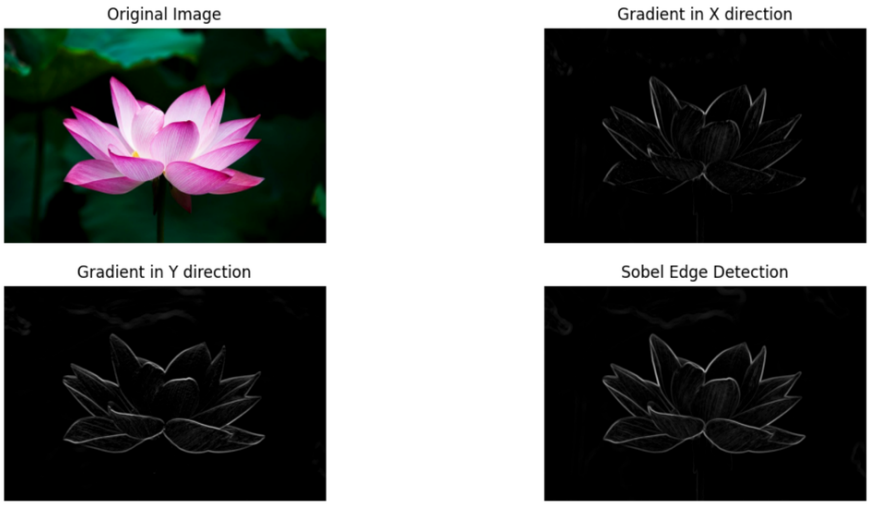

Fig. 2: Example of Sobel edge detection[12]

4 Canny Edge Detector

The canny edge detector takes the output of the Sobel operator and refines to give the clearest edges. It does this in two steps, non-maximum suppression and hysteresis thresholding.

4.1 Non-maximum suppression

Depending on the resolution of the original image, the thickness of the edges can vary. Non-maximum suppression finds thick edges and keeps only their local maxima. In other words, it takes the edge with the highest intensity and removes the parallel edges.

The second step, hysteresis thresholding has already been described in I.4.2. With all of these steps combined the canny edge detector is able to give the edges with the least amount of noise.

Fig. 3: Example of Canny edge detection[12]

II. Laplacian Methods

The method for edge detection is classified into two categories: gradient-based and Laplacian-based. This section focuses on the latter.

In the Laplacian-based method, an image is used to compute the second-order derivative expression which has a zero crossing. Generally, edges are found by searching for a zero crossing of a non-linear differential expression. Typically, a pre-processing step of Gaussian smoothing is applied for edge detection, which is commonly a refining stage. [5, 14]

5 The Laplacian Operator

The Laplacian operator is a second-order differential operator defined as the divergence of the gradient of a function. In the context of image processing, it measures the rate at which the average value of a pixel's neighbourhood deviates from the pixel itself.

5.1 Gradient and Divergence

The gradient of a function $f(x, y)$ is a vector that points in the direction of the greatest rate of increase of the function. For an image, the gradient is given by:

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right)$$

The divergence of a vector field $\mathbf{F} = (F_x, F_y)$ is a scalar that represents the rate at which "density" exits a point. It is given by:

$$\nabla \cdot \mathbf{F} = \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y}$$

5.2 Laplacian Operator

The Laplacian operator is the divergence of the gradient of a function. For a function $f(x, y)$, the Laplacian Δf is given by:

$$\Delta f = \nabla \cdot (\nabla f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

In a discrete 2D image, the second derivatives can be approximated using finite differences. The commonly used discrete Laplacian kernel is:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

6 Marr-Hildreth Edge Detector

More commonly known as the Laplacian of Gaussian (LoG) edge detector, it uses Gaussian smoothing and zero-crossing further to enhance the accuracy of the general Laplacian operator. [14]

Laplacian of Gaussian is very useful because the 2nd derivative is very sensitive to noise, and this is helpful in filtering noise from the image.

6.1 Gaussian Smoothing

The image is first smoothed using a Gaussian filter. This helps to reduce noise before detecting edges, as high-frequency noise could create false edges. The Gaussian filter is given by:

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

where σ is the standard deviation, controlling the amount of smoothing.

6.2 LoG Function

The LoG function combines Gaussian smoothing with the Laplacian operator. It is used to detect edges by finding the zero crossings of the second derivative after smoothing the image with a Gaussian filter.

6.3 2-D LoG Equation

Laplacian of Gaussian (LoG) Computation: After smoothing the image, the Laplacian of Gaussian is computed. The Laplacian is a second-order derivative, which highlights areas of rapid intensity change (edges). [13, 2]

The LoG function is the combination of the Laplacian and Gaussian:

$$\mathrm{LoG}(x,y) = \nabla^2 \left(G(x,y)\right) = \frac{1}{\pi\sigma^4}\left(1 - \frac{x^2+y^2}{2\sigma^2}\right) e^{-\frac{x^2+y^2}{2\sigma^2}}$$

This function is often referred to as the Mexican hat because of its shape when plotted in 3D, which resembles a hat with a raised center and steep edges. [9]

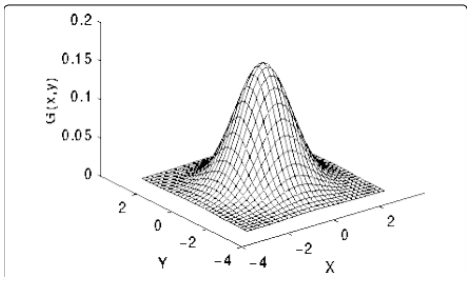

Fig. 4: "Mexican-hat" function

6.4 Zero Crossing Detection

The edges in the image are found by detecting the zero crossings of the LoG function. A zero crossing occurs when the second derivative (Laplacian) changes sign, which often corresponds to the presence of an edge. This can be achieved by scanning the image and checking where the value of the Laplacian changes from positive to negative or vice versa. These locations are where significant changes in intensity occur, and they are considered edges. By focusing on the zero crossings of the second derivative, we can filter out less significant edges, which might be detected by the first derivative alone. [5, 4, 6]

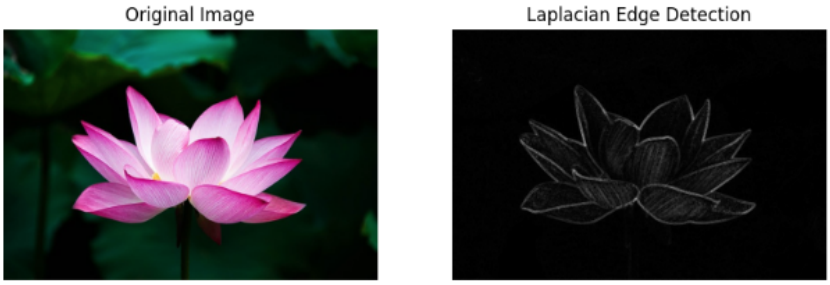

Fig. 5: Example of LoG edge detection [12]

7 Breadth-First Search Edge Detection (Bonus)

In this novel algorithm, we represent a gray-scale image as a graph, with each pixel as a node. Each node is connected to its neighboring pixels in the four cardinal directions: top, bottom, left, and right.

7.1 The Algorithm

We present a pseudocode and later, a python code for the algorithm.

Algorithm 1 Edge Detection using BFS

Input: Image I, Threshold T, Starting Point P
Output: List of detected edges
Initialize an empty queue Q
Initialize an empty list for detected edges *Edges*
Enqueue the starting point P to Q
Mark point P as visited
while Q is not empty **do**
    Dequeue a point *current* from Q
    for each neighbor n of *current* **do**
        if n is not visited **then**
            if $\text{abs}(I[current] - I[n]) > T$ **then**
                Add n to *Edges*
                Enqueue n to Q
            end if
            Mark n as visited
        end if
    end for
end while
return *Edges*

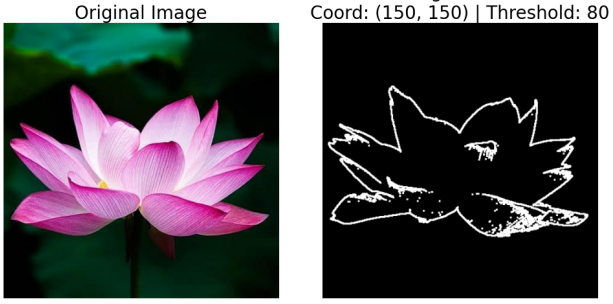

Fig. 6: Example of BFS edge detection

7.2 Breadth-First Search (BFS) for Edge Detection

Starting from any given point in the image, we perform a breadth-first search (BFS) until we reach a node where the brightness value differs by a fixed threshold value. This helps in identifying the boundary or edge of an object. The algorithm's performance depends on two factors:

- **Starting Point**: If the starting point lies inside an object, the algorithm will trace all the details of the interior of the shape. Conversely, if the starting point is outside, it will yield a clear silhouette of all objects present. Starting in any corner of the image (ideally at (0,0)) will lead to a silhouette map of all shapes in the image. See rows of Figure 7

- **Threshold** : The lower the threshold, the more sensitive is the algorithm to detecting edges, and will give much more detail for an open curve. However, this will come at the cost of random noise also being detected as some edges. A higher threshold value is better for detecting solid boundaries like silhouettes. See columns of Figure 7

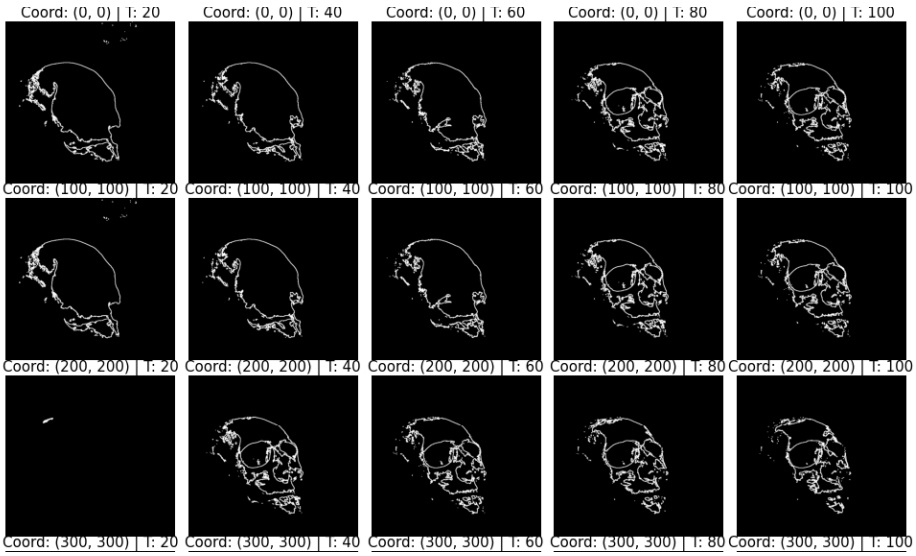

Fig. 7: Different detections based on thresholds and starting coordinates

7.3 Advantages of the Algorithm

This BFS-based edge detection algorithm has several notable advantages that make it particularly effective for various image processing tasks:

7.3.1 Handling Discontinuous Edges

The algorithm is especially useful for images that contain edges with discontinuities or gaps. Traditional edge detection methods often fail in the presence of such interruptions. However, since BFS explores pixels layer by layer, it is capable of detecting edges even when they are not continuous. It can "sneak through" small gaps in the edges and still detect the overall boundary, making it particularly effective in images where edges may be fragmented or incomplete, such as in hand-drawn sketches or imperfect object contours. Refer to Figure 8

7.3.2 Scalability for Colored Images

While the current version of the algorithm is designed for grayscale images, it can be easily expanded to work with colored images. In colored images, the threshold for detecting edges can be adjusted dynamically based on parameters such as the difference in color channels (e.g., RGB) or pixel intensity. This means that the algorithm can be fine-tuned to detect edges more accurately by considering the variation in color between adjacent pixels. This flexibility makes the algorithm adaptable to a wider range of image types, from simple black-and-white drawings to more complex colored images, while maintaining its effectiveness.

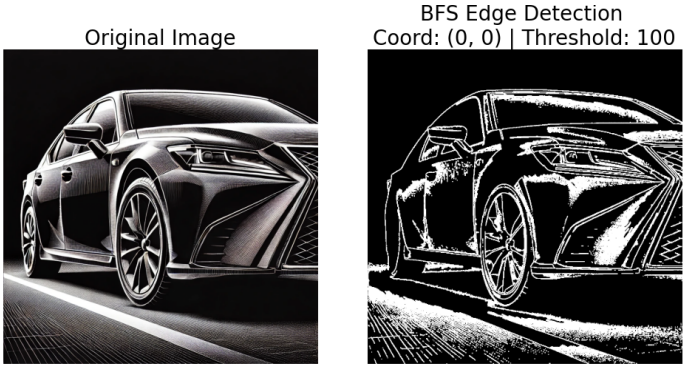

Fig. 8: BFS Edge detection for imperfect boundaries

7.3.3 Targeted Edge Detection for Specific Objects

If a user selects a starting point inside a specific object, the algorithm can effectively map the edges of that object without affecting the rest of the image. This is particularly advantageous when you are only interested in detecting the edges of one object in the image, rather than performing a global edge detection across the entire scene. By focusing only on a single object, the algorithm reduces computational cost, as it avoids the unnecessary processing of pixels outside the region of interest. This targeted approach enhances both speed and efficiency when dealing with complex images with multiple objects. See Figure 9

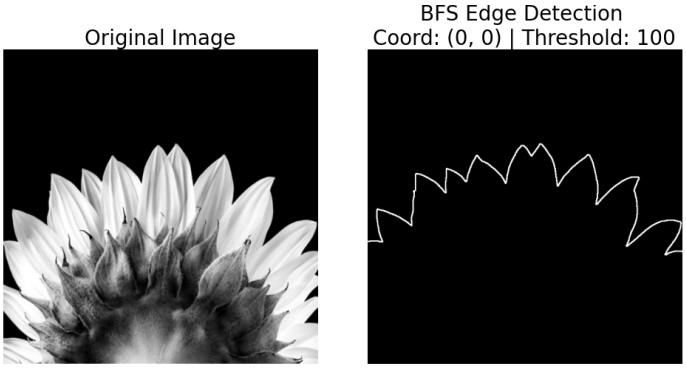

Fig. 9: Finding only the boundary

7.4 Disadvantages of the Algorithm

While the BFS-based edge detection algorithm offers several advantages, it also comes with certain limitations that need to be considered:

7.4.1 Sensitive to Threshold Value

The performance of the algorithm is heavily dependent on the threshold value used for edge detection. If the threshold is too low, the algorithm may detect too many spurious edges, leading to noise and false positives. Conversely, if the threshold is too high, the algorithm may miss subtle edges or fail to detect any edges at all. Finding the optimal threshold can be a challenge and may require fine-tuning based on the image content, making the algorithm sensitive to parameter settings. Notice how some edge detections are very accurate while some appear meaningless for the same coordinates in Fig. 10

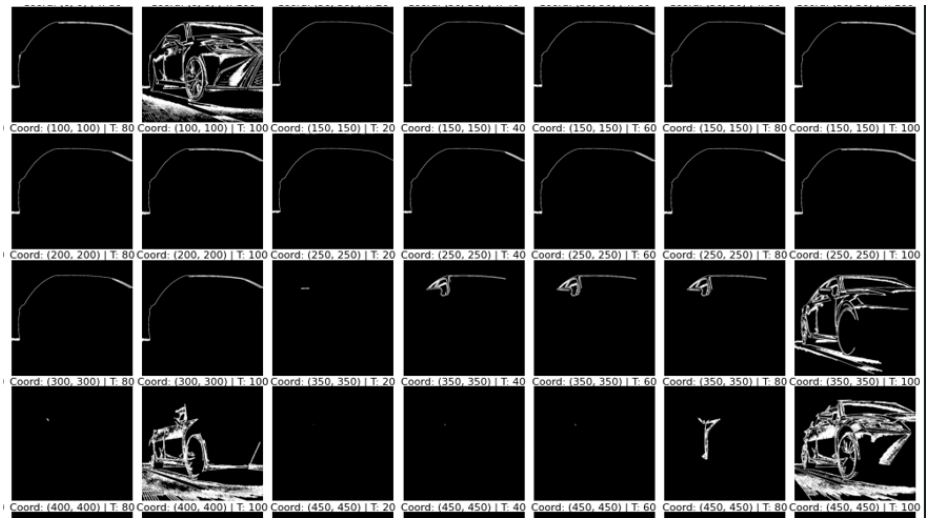

Fig. 10: Accuracies and inaccuracies based on threshold values

7.4.2 Limited to Local Edge Detection

The BFS approach detects edges based on pixel differences from a specific starting point, which means it can only detect edges in the immediate vicinity of the starting point. This may lead to incomplete edge maps, especially in large or complex images. If the starting point is poorly chosen or if there are multiple disconnected regions in the image, the BFS algorithm might fail to detect all relevant edges across the entire image. As a result, it might require multiple runs from different starting points or additional processing steps to complete edge detection for the entire image. Notice how it only detects portions in Figure 14

7.4.3 Computational Cost for Large Images

Although the algorithm is efficient in targeted edge detection (i.e., focusing on a specific object), it can still be computationally expensive for large images with high resolution or many complex structures. Since BFS explores each pixel and its neighbors, the time complexity increases with the size of the image. For large images, this can lead to performance issues, especially if multiple edges or regions need to be processed. In such cases, the algorithm may require significant computational resources and time to complete.

7.4.4 Not Robust to Noise

The algorithm's reliance on pixel differences can make it susceptible to noise in the image. In noisy environments, small variations in pixel values can be mistaken for edges, leading to false positives and unnecessary edge detections. Preprocessing steps such as smoothing or denoising might be necessary before applying the algorithm to noisy images, adding extra complexity to the overall process. Compare and contrast Fig 11 and Fig 12. However increasing the threshold may fix the noisy edges introduced in this case.

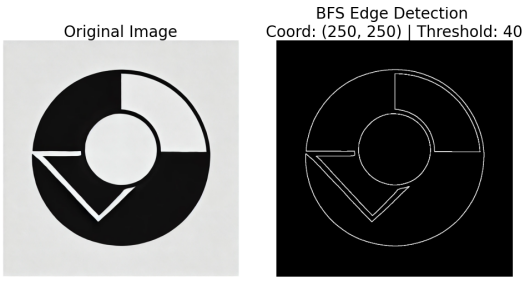

Fig. 11: Simple image with no noise

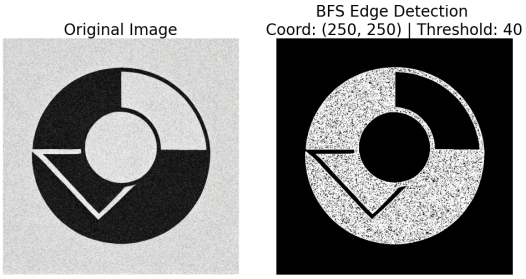

Fig. 12: Adding noise to Fig 11

7.4.5 Limited Flexibility in Complex Scenes

For images with complex textures, overlapping objects, or intricate details, the BFS algorithm may struggle to detect edges accurately. Since it works by iterating through the image based on a starting point, it may fail to distinguish between important edges and noise or artifacts in highly detailed or cluttered images. This makes the algorithm less suitable for scenes where precise edge detection is required in highly complex environments. See Figure 14

Fig. 13: Original Image for Fig. 14

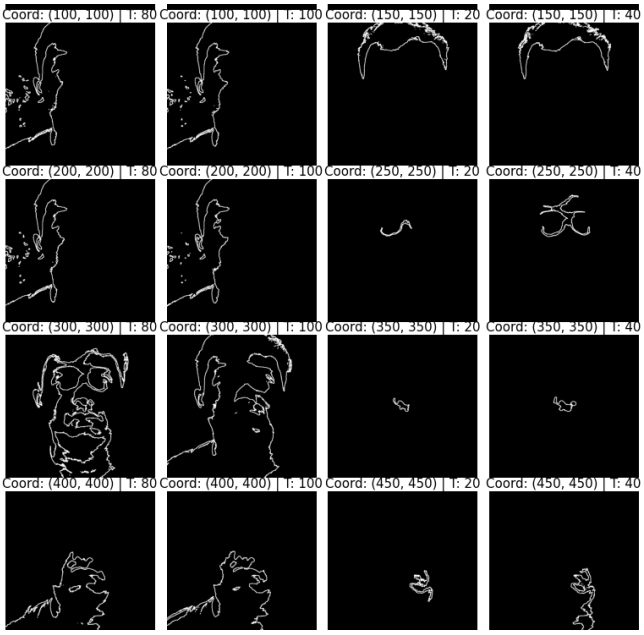

Fig. 14: BFS edge detection fails for a complex image of a face

8 Python Code for BFS-based Edge Detection

Linked is the Python code that implements a BFS-based edge detection algorithm. Use the `generate_image` function to generate a single edge image based on given parameters, and the `generate_image_grid` function for a grid of edge images for a pre-set combination of threshold and start coordinates.

GitHub Repository for BFS-based Edge Detection

9 Further Research

1. The use of fuzzy logic [10] is a non-classical approach to the problem of edge detection. Fuzzy logic edge detection is a method that leverages the principles of fuzzy logic to detect edges in an image. Unlike traditional edge detection techniques, which make binary decisions about whether a pixel is part of an edge, fuzzy logic edge detection handles the uncertainty and partial truth, making it more robust in dealing with noisy and blurred images.

2. Tuning the threshold hyper parameter using statistical machine learning tools like linear regression.

References

[1] Girish N. Chaple, R. D. Daruwala, and Manoj S. Gofane. "Comparisons of Robert, Prewitt, Sobel operator based edge detection methods for real time uses on FPGA". In: *2015 International Conference on Technologies for Sustainable Development (ICTSD)*. 2015, pp. 1–4. DOI: 10.1109/ICTSD.2015.7095920.

[2] E.R. Davies. "Constraints on the design of template masks for edge detection". In: *Pattern Recognition Letters* 4.2 (1986), pp. 111–120.

[3] et al Dr. Mike Pound Brady John Haran. *Computer Vision: Filters (Blur, Edge Detection etc)*. 2015. URL: https://www.youtube.com/playlist?list=PLzH6n4zXuckoRdljSlM2k35BufTYXNNeF.

[4] W. Frei and C.C. Chen. "Fast boundary detection: A generalization and a new algorithm". In: *IEEE Trans. Pattern Analysis and Machine Intelligence* 26.4 (1977), pp. 988–998.

[5] *Gradient Based Edge Detection*.

[6] R.M. Haralick. "Digital step edges from zero crossing of the second directional derivatives". In: *IEEE Trans. Pattern Analysis and Machine Intelligence* 6.1 (1984), pp. 58–68.

[7] N. Kanopoulos, N. Vasanthavada, and R.L. Baker. "Design of an image edge detection filter using the Sobel operator". In: *IEEE Journal of Solid-State Circuits* 23.2 (1988), pp. 358–367. DOI: 10.1109/4.996.

[8] Jamie Ludwig. *Image Convolution*. URL: https://web.pdx.edu/~jduh/courses/Archive/geog481w07/Students/Ludwig_ImageConvolution.pdf.

[9] E. Nadernejad, S. Sharifzadeh, and H. Hassanpour. "Edge detection techniques: Evaluations and comparisons". In: *Applied Mathematical Sciences* 2.31 (2008), pp. 1507–1520.

[10] D. D. Nawgaje, Rajendra, and D. Kanphade. "Implementation of fuzzy logic for detection of suspicious masses in mammograms using DSP TMS320C6711". In: *International Journal of Advanced Engineering and Application* (2011).

[11] Shree Nayar. *Edge Detection Using Gradients | Edge Detection*. 2021. URL: https://www.youtube.com/watch?v=lOEBsQodtEQ&list=PL2zRqk16wsdqXEMpHrc4Qnb5rA1Cylrhx&index=3.

[12] Roboflow. *Edge Detection*. Accessed: 2024-11-17. 2024. URL: https://blog.roboflow.com/edge-detection/.

[13] K.A. Stevens. *Surface perception from local analysis of texture and contour*. Tech. rep. AI-TR-512. Cambridge, MA: M.I.T. Artificial Intelligence Laboratory, 1980.

[14] V. Torre and T.A. Poggio. "On edge detection". In: *IEEE Trans. Pattern Analysis and Machine Intelligence* 8.2 (1986), pp. 163–187.