Journal of Dynamical Systems and Complexity (2026) vol.1(2) 1-14
eISSN: 3122-4326, DOI: 10.66131/JDSC1220261-14

Performance optimization of edge detection algorithms using ABC-ED metaheuristics and SIMD Instructions

Elizabeth Martínez[1]*, Eduardo Rodríguez[1], Eusebio Bugarin[1], Ana Aguilar[1]

[1] Departamento de Eléctrica y Electrónica, Tecnológico Nacional de México/IT de Ensenada, Baja California, México

Abstract

Edge detection plays a crucial role in identifying boundaries and key features in images, providing valuable information for various applications such as object recognition, image segmentation, pattern analysis, and stereoscopic vision. However, this process, often required in real-time, involves a high computational demand, which has driven significant efforts toward optimization. This article compares two performance-oriented approaches to efficiently address this task: a method based on SIMD (Single Instruction, Multiple Data) instructions, and a modified version of the Artificial Bee Colony metaheuristic for edge detection (ABC-ED). First, the SIMD instruction method is implemented, performing parallel edge inspection by simultaneously processing four pixels and optimizing resource usage. Next, the ABC-ED method is developed, highlighting its effectiveness in detecting edges by evaluating less than one-third of the image's pixels. Nevertheless, despite narrowing the search region, it still exhibits a high execution time. In contrast, the SIMD instruction method demonstrates efficiency and speed in edge detection, positioning it as the most suitable option for this task. A comparison with other state-of-the-art methods reveals superior performance of the SIMD instruction method, even when compared to implementations in the well-known OpenCV open-source library aimed at the same task.

Keywords— Edge detection, artificial bee colony method, ABC-ED, SIMD instructions, optimization.

1 Introduction

Edge detection in images is an essential component in the field of image processing and computer vision [1]. This process plays a critical role in enabling the identification of significant transitions in pixel intensity [2], which is leveraged in various applications such as object recognition, image segmentation, pattern analysis, and stereoscopic vision [3]. However, a significant challenge faced by these algorithms is their high computational load due to the intensive image processing operations they involve, such as convolutions and high-precision filtering to highlight abrupt changes in pixel intensity levels. These operations consume significant computational resources, resulting in high execution times, which affect real-time applications where an instant response is required.

To address this challenge, various strategies and techniques have been proposed over time. One notable approach is the implementation of algorithms on Graphics Processing Units (GPUs), which have proven effective by leveraging the GPU's ability to execute tasks in parallel [4]. Other important considerations include the use of Convolutional Neural Networks (CNNs) designed for edge detection [5].

Additionally, metaheuristic approaches such as evolutionary algorithms and Artificial Bee Colony (ABC) algorithms have been explored to optimize the parameters of edge detection algorithms [6, 7]. Some have modified the Artificial Bee Colony method to optimize edge searching in the image without the need to evaluate all its pixels (ABC-ED) [8, 9].

Furthermore, strategies like parallelization and the use of SIMD instructions have been considered [10, 11]. Parallelization involves dividing the data into smaller parts to process them simultaneously across different cores or processing units. On the other hand, SIMD instructions allow performing arithmetic or logical operations in parallel on multiple data elements in a single instruction, especially benefiting algorithms that involve repetitive operations on pixels. In this context, [12] proposed a method to accelerate edge detection by reducing redundant arithmetic operations and leveraging data parallelism via AVX instructions on multi-core CPUs. They demonstrated that this purely software-based strategy outperforms standard OpenCV implementations without requiring dedicated hardware accelerators.

*Corresponding author: elimm1910@gmail.com
Received: July 29, 2025, **Published:** Jun 30, 2026
Editor: L. G. de la Fraga

This article presents the implementation and evaluation of two specific strategies: the SIMD instruction method and the ABC-ED method. The SIMD approach was designed with the capability to simultaneously traverse four pixels of the image, executing the convolution in parallel for efficient edge identification, resulting in significant processor resource optimization. On the other hand, the ABC-ED method integrates parallel convolution and is optimized in terms of memory access, data manipulation, and redundancy elimination. This method is capable of obtaining the representation of the edges of the image, evaluating only 20% of the image in a time superior to that proposed by [8].

This article is structured into six sections: after this introduction, the fundamental procedures for edge detection are explained in section two. Section three presents the implementation of the SIMD instruction method, followed by section four, which explains the structure of the ABC-ED method. Section five presents the obtained results, and the conclusion follows in section six.

2 Edge Detection

Edge detection is the initial step in numerous computer vision applications [8]. Its function is to significantly reduce the amount of information to be processed by filtering out irrelevant data while preserving the key structural features of an image. The primary purpose of this process is to identify and locate significant changes in image properties, such as changes in color intensity, texture, or depth, which suggest possible boundaries between objects or regions in the image.

There are several methods to perform edge detection, which can generally be classified into two categories: Gradient methods based on the first-order derivative, and Laplacian methods based on the second-order derivative [8, 1, 2]. Additionally, there is the Canny method, which combines strategies from both categories [3]. In this case, we focus on one of the Gradient methods called the Sobel operator, which is based on convolving the image with a small kernel that consists of two two-dimensional convolution masks, one for detecting horizontal changes and the other for detecting vertical changes. The horizontal Sobel kernel (G_x) and the vertical Sobel kernel (G_y) are 3x3 matrices used to calculate partial derivatives in the horizontal and vertical directions, respectively. These kernels are:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}; G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}. \tag{1}$$

The convolution of the original image with the horizontal and vertical Sobel kernels is performed by applying the masks G_x and G_y to each pixel of the image and its neighborhood. This is done by multiplying each element of the mask with the corresponding pixel value and summing the results. The gradient magnitude is then computed by combining the responses obtained from the horizontal and vertical Sobel kernels. For each pixel (x, y), the gradient magnitude G is calculated according to:

$$G = \sqrt{G_x^2 + G_y^2}. \tag{2}$$

Finally, the result of the Sobel operator can be thresholded to highlight the pixels that correspond to edges. This involves setting a threshold and classifying the pixels based on whether their gradient magnitude exceeds that threshold.

3 SIMD Instruction Method

SIMD instructions are a set of computing instructions designed to perform parallel operations on data sets [13], significantly accelerating data processing in applications that require performing the same operation on multiple data elements simultaneously. This parallel processing approach has become a fundamental tool in modern computing, enhancing the performance and efficiency of a wide range of applications, such as image processing, signal processing, scientific simulation, machine learning, and more.

The idea behind SIMD instructions is to execute a single instruction across multiple data elements at the same time, reducing the processor's workload and speeding up the execution of data-intensive operations. To store multiple data elements in parallel, SIMD registers are used, which uniquely treat each element independently during the execution of SIMD instructions [14]. In this context, SIMD operations perform identical functions on multiple data elements in a single clock cycle. For instance, when four values are held in a SIMD register, the SIMD instruction performs the same operation simultaneously on all four values. This approach is particularly effective in situations

where identical or similar operations are carried out on large data sets, such as image or video processing. By using SIMD instructions, developers can maximize the processing capacity of modern processors and achieve optimal performance in their applications.

Over the past few decades, SIMD instructions have evolved and become increasingly sophisticated, with the introduction of extensions such as SSE (Streaming SIMD Extensions), AVX (Advanced Vector Extensions), and NEON (on ARM architectures) [13], providing broader instruction sets and enhanced parallel processing capabilities. These extensions have allowed developers to perform even more complex operations in parallel, significantly contributing to the development of high-performance applications across various fields. In this work, SSE is chosen since we are working with an Intel microprocessor, ensuring the corresponding compatibility and optimization. To use SSE, we employ the `<immintrin.h>` library, which provides the functions and macros to use SIMD intrinsic instructions in x86 and x86-64 architectures, such as SSE and AVX.

In the field of computational image processing, SIMD instructions offer substantial advantages [15, 16]. Operations such as convolutions, matrix multiplications, and pixel manipulation benefit significantly from the parallel processing capabilities of SIMD instructions. In convolution operations, for example, SIMD can apply the filter to multiple pixels at once, significantly reducing the processing time compared to traditional methods where each pixel is processed sequentially. Similarly, in matrix multiplications, which are common in tasks such as image transformation or machine learning, SIMD can handle multiple rows or columns in parallel, reducing computation time and improving efficiency. Pixel manipulation tasks such as color adjustments or applying filters can also take advantage of SIMD by modifying multiple pixel values in a single operation, speeding up real-time processing in imaging or video applications.

However, despite the clear performance improvements that SIMD can bring, not all processes can be effectively parallelized. Certain algorithms require operations to be performed sequentially because the result of processing a pixel may depend on the values of its neighbors or on previous calculations. In these cases, SIMD cannot be applied directly, as the parallel nature of SIMD instructions depends on the independence of each operation on data elements. Furthermore, implementing SIMD effectively is not an easy task. It requires a deep understanding of the underlying processor architecture, including how data is aligned in memory, how SIMD registers are used, and how different SIMD instruction sets are optimized for various operations, making it challenging to integrate into more complex processes.

To accelerate edge detection runtime, we avoid processing each pixel individually and instead process the image in blocks of four pixels. This is made possible by the potential of SSE instructions, which allow storing a total of four 32-bit floating-point numbers in their 128-bit registers, enabling operations on these four values in a single clock cycle, as illustrated in Figure 1.

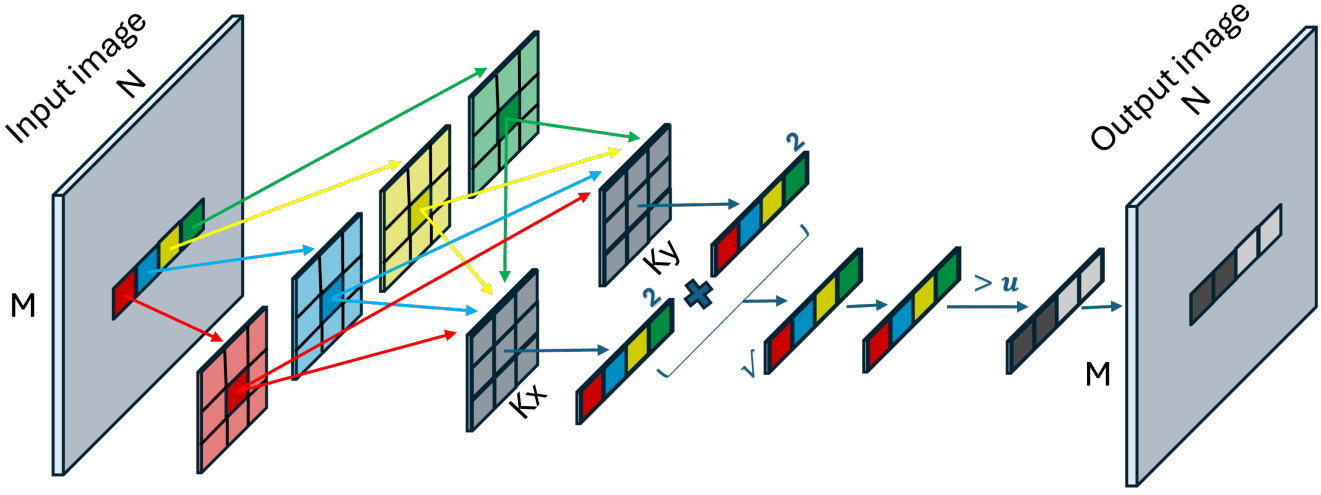

Figure 1: Representation of the edge detection process using SIMD instructions.

For better understanding of the procedure, the pseudocode is presented in Algorithm 1. The outer loop iterates over the rows of the image, while the inner loop, which increments in SIMD steps, processes the pixels in each row simultaneously. Parallel convolution can then be applied as shown in Algorithm 2.

Algorithm 1 Edge detection with SIMD Steps.

```
Input:
    input → Input image
    u → Threshold value
    kernelx, kernely → Sobel kernels
    kernelSize → kernel dimensions (=3)
Output: Binarized output image with edges
process: Edge Detection SIMD
    simd_step ← 4
    kernelRadius ← kernelSize / 2
    width_simd ← input.width - kernelSize
    height_simd ← input.height - kernelSize
    for y from 1 to height_simd do
        for x from 1 to width_simd step simd_step do
            resultx, resulty ← ParallelSSEConvolution(input, kernelx, kernely, x, y, kernelSize, kernelRadius)
            resultxSquared ← Mul(resultx, resultx)
            resultySquared ← Mul(resulty, resulty)
            sumOfSquares ← Add(resultxSquared, resultySquared)
            magnitude ← Sqrt(sumOfSquares)
            threshold ← Set1(u)
            compare ← Cmpgt(magnitude, threshold)
            edge ← And(compare, Set1(255))
            Storeu(binarizedImage(x, y), edge)
        end for
    end for
end process
```

Algorithm 2 Parallel SIMD Sobel Convolution

```
Input:
    input → Input image
    kernelx, kernely → Sobel kernels
    x, y → Current pixel coordinates
    kernelSize → Kernel size (=3)
    kernelRadius → Kernel radius
Output:
    resultx, resulty → Horizontal and vertical gradient vector
Process: Parallel SIMD Convolution
    resultx ← Setzero()
    resulty ← Setzero()
    for ky from −kernelRadius to kernelRadius do
        for kx from −kernelRadius to kernelRadius do
            srcX ← x + kx
            srcY ← y + ky
            inputPixels ← Loadu(&input[srcX, srcY])
            kernelValuex ← Set1(kernelx[(ky + kernelRadius) * kernelSize + (kx + kernelRadius)])
            kernelValuey ← Set1(kernely[(ky + kernelRadius) * kernelSize + (kx + kernelRadius)])
            resultx ← Add(resultx, Mul(inputPixels, kernelValuex))
            resulty ← Add(resulty, Mul(inputPixels, kernelValuey))
        end for
    end for
return resultx, resulty
end process
```

The convolution is performed in two directions, horizontal (on the x-axis) and vertical (on the y-axis), using the convolution kernels specified by "kernelx" and "kernely". The process is efficiently carried out using 128-bit SIMD registers (`__m128`), allowing simultaneous operations on multiple data elements. A set of SIMD instructions (see Table 1) is then used to load the input pixels and kernel values into SIMD registers, perform parallel convolution, and accumulate the sum of the results.

Table 1: SIMD Instructions.

Instruction	Operation
_mm_setzero_ps	Initialize a 128-bit register with four floating-point values set to zero
_mm_loadu_ps	Load four unaligned floating-point values from memory into a 128-bit register
_mm_set1_ps	Create a 128-bit register with all elements initialized to the same floating-point value
_mm_add_ps	Perform parallel floating-point addition on two 128-bit registers
_mm_sqrt_ps	Perform parallel floating-point square root operations on the elements of a 128-bit register
_mm_storeu_ps	Store a 128-bit register with floating-point values into an unaligned memory location
_mm_cmpgt_ps	Perform a "greater than" comparison between the values stored in two 128-bit registers

After performing the convolution, additional operations are carried out to obtain the absolute values of the derivatives in both directions, sum the squares of these derivatives, and finally compute the square root of the result. These calculations are part of the edge detection process.

The final results are compared to the established threshold using the instruction (`__mm__cmpgt_ps`). If the calculated value exceeds the threshold "u", the maximum value of 255 is assigned, indicating the presence of an edge. Otherwise, the value is set to 0. Finally, the obtained values are stored in the corresponding position in the binarized image.

4 ABC-ED Method

The Artificial Bee Colony (ABC) Algorithm is a metaheuristic optimization algorithm first conceived by Karaboga in 2005 [8]. This computational approach is inspired by the behavior of honeybees during the search and collection of food sources. The core of the algorithm lies in its ability to emulate the collective intelligence inherent in bee colonies. Just as honeybees share information about the location of successful food sources through dances and chemical signals, the algorithm uses a distributed approach where multiple virtual agents collaborate to explore and exploit solutions in a search space. This cooperation between virtual individuals leads to more efficient exploration and the identification of optimal solutions in combinatorial optimization problems. To simulate this behavior in an algorithmic model, three types of bees are simulated:

- Scout Bees: These bees randomly search the solution space for candidate solutions. Their goal is to find new promising areas.
- Employed Bees: Employed bees exploit known food sources, improving the quality of local solutions. They make adjustments and enhancements to existing solutions.
- Onlooker Bees: Onlooker bees are responsible for selecting food sources based on the information provided by the employed bees. They may probabilistically choose food sources and communicate their selection to other bees.

The ABC-ED algorithm [8] is a modification of the ABC algorithm aimed at efficient localization in edge detection in grayscale digital images, integrating the classic Sobel method for identification with the ABC algorithm for localization. This method demonstrates the ability to obtain a representation of the image's edges while exploring less than one-third of it. Despite this achievement, the proposed execution time remains considerable. To address this aspect, an optimized implementation of the algorithm has been developed to evaluate and improve its performance.

In general, the algorithm is based on the high probability that at least one of the neighboring pixels of an edge is also an edge. For this reason, it first identifies a set of pixels that correspond to edges and then inspects their

neighborhoods to find new edges while also stochastically searching in new random regions of the image to ensure that it generalizes across the entire image. To accomplish this, the algorithm consists of five fundamental phases, each aimed at simulating the behavior of honeybees, as shown in Figure 2.

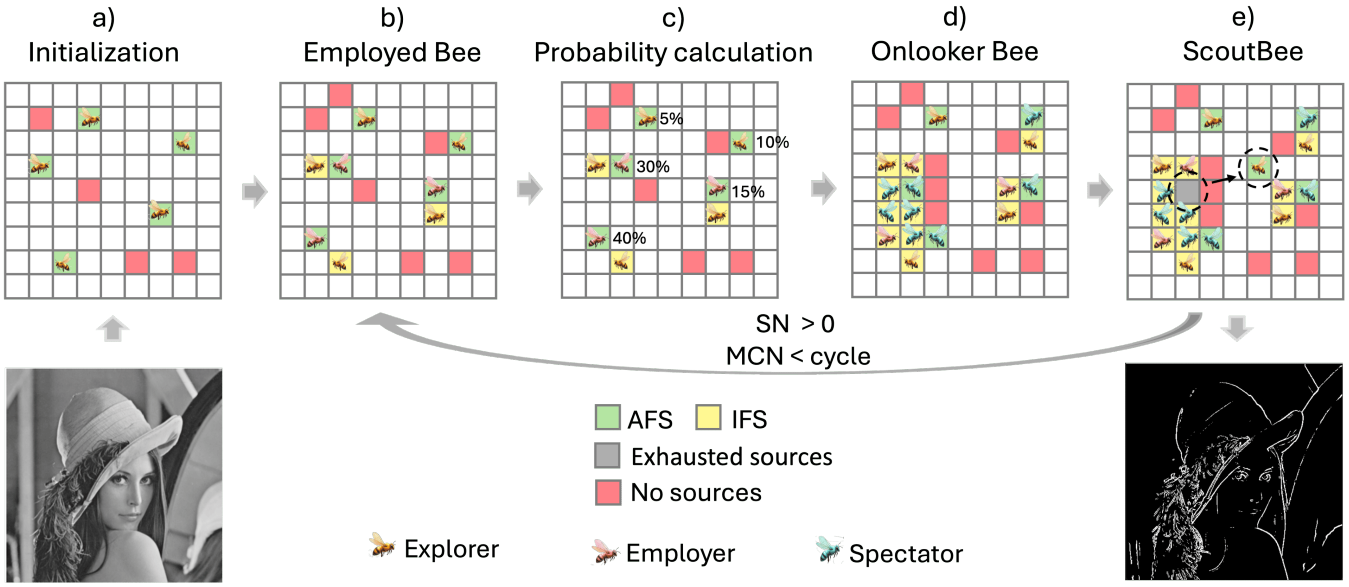

Figure 2: Representation of the edge detection process using the ABC-ED method.

The first phase is initialization, where a set of food sources is created. In this case, a food source represents a pixel in the image that corresponds to an edge (see Figure 2a). To find this initial set of food sources, several pixels in the image are randomly inspected until SN food sources are obtained. SN is one of the control parameters calculated according to:

$$SN = \sqrt{width * height}. \tag{3}$$

Where "*width*" and "*height*" refer to the width and height of the image, respectively.

In the second phase, known as Employed Bees, a random neighbor of each active source is selected. If this neighbor turns out to be an edge, it replaces the current source, and the original source becomes inactive, as shown in Algorithm 3. This phase also accounts for the possibility that the selected neighbor has already been analyzed and is now an inactive source, during the execution of the *GetNeighborCandidate* function explained in Algorithm 6. In such cases, the current source is still replaced by the neighbor, using its previously stored parameters, such as magnitude and neighborhood information.

In the third stage, the probability of each active source is calculated using:

$$P(fk) = \frac{fit(fk)}{\sum_{i=1}^{SN} fit(fk)}. \tag{4}$$

Where "fk" represents the source under analysis, and "$fit(fk)$" refers to the quality of the source, which in this case is the magnitude computed using the Sobel operator. This probability is then used in the fourth phase, the Onlooker Bees stage, where a roulette wheel selection mechanism is created [9]. Sources with a higher likelihood of having neighboring edges are more likely to be selected. For the selected sources, the same process as in the Employed Bees phase is applied, as illustrated in Algorithm 4. When a source is replaced by a neighbor in this phase, it inherits the probability of the original source, ensuring its place in the selection roulette is maintained.

Algorithm 3 Employed Bees Phase.

Input:
AFS → Active Food Sources
IFS → Inactive Food Sources
u → Threshold
image → Input image
output_image → Binary output image
count_inspections → Inspected pixel counter
Output:
Updated AFS, IFS and output_image
Process: Employed Bee Exploration
for $i \leftarrow 0$ **to** $|AFS| - 1$ **do**
fk ← AFS[i]
nfk ← GetNeighborCandidate(fk, u, AFS, IFS, image, output_image, count_inspections)
if nfk.fit > u **then**
ReplaceFoodSource(fk, nfk, AFS, IFS, i)
end if
end for
end process

Algorithm 4 Onlooker Bees Phase.

Input:
AFS → Active Food Sources
IFS → Inactive Food Sources
SN → Number of food sources
u → Threshold
image → Input image
output_image → Binary output image
count_inspections → Inspected pixel counter
Output:
Updated AFS, IFS and output_image
Process: Onlooker Bee Selection
t ← 0
while $t < SN$ **do**
r ← Random(0,1)
for $i \leftarrow 0$ **to** $|AFS| - 1$ **do**
if $r \in fk.range$ **then**
if fk.neighbors_by_chosen[8] $\neq 0$ **then**
t ← t + 1
nfk ← GetNeighborCandidate(fk, u, AFS, IFS, image, output_image, count_inspections)
if nfk.fit > u **then**
ReplaceFoodSource(fk, nfk, AFS, IFS, i)
end if
end if
break
end if
end for
end while
end process

In the final phase, known as the Scout Bee phase, the active sources are evaluated to determine if they are exhausted. If a source is found to be exhausted, it is replaced by another source. This replacement can either be one of the sources that became inactive after being replaced by a neighbor, or it can be a newly discovered source

randomly located in the image (refer to Algorithm 5). The choice between these two options is controlled by the parameter "ε", which is expressed as a percentage. A higher value of "ε" increases the likelihood that the replacement will be a new source, while a lower value favors replacing it with an inactive source. In this case, "ε" is set to 50.

Algorithm 5 Scout Bee Phase.

```
Input:
    AFS → Active Food Sources
    IFS → Inactive Food Sources
    SN → Number of food sources
    u → Threshold
    limit, e → Replacement control parameters
    image → Input image
    output_image → Binary output image
    count_inspections → Inspected pixel counter
Output:
    Updated AFS, IFS, and output_image
Process: Scout Bee Exploration
    for i ← 0 to |AFS| − 1 do
        if fk.neighbors_by_chosen[8] = 0 then
            while true do
                fr ← New Source()
                replacement_form ← GetReplacementForm(IFS, e, count_inspections)
                if replacement_form = 0 then
                    GetNewFoodSource(SN, u, AFS, IFS, image, output_image, fr, count_inspections)
                    if fr.fk ≠ (0,0) then
                        AFS[i] ← fr
                        output_image[fr.fk.y, fr.fk.x] ← 255
                        break
                    end if
                end if
                if replacement_form = 1 then
                    if not IFS.empty() then
                        fr ← FirstElement(IFS)
                        RemoveFirstElement(IFS)
                        AFS[i] ← fr
                        break
                    end if
                end if
                if replacement_form = 2 then
                    SN ← 0
                    break
                end if
            end while
        end if
    end for
end process
```

Except for the Initialization phase, the other phases are repeated a set number of times based on the maximum number of cycles established (MCN). This parameter determines the relationship between efficiency and speed. As the number of cycles performed increases, the number of edges detected by the algorithm also increases; however, this also raises the execution time.

To organize food sources according to their status, two data sets have been established: AFS, intended for active sources, and IFS, reserved for inactive sources that have not yet been fully explored. To optimize performance, the `unordered_set` data type has been used in the IFS set, this is a C++ standard library class that implements a container for unordered sets using a hash table. This type of container provides constant time average performance

for find, insert, and delete operations, regardless of the size of the input data. This feature allows you to perform actions on the elements of these sets without requiring a complete walkthrough of the set. In the case of the AFS set, the `vector` data type has been chosen, another class from the C++ standard library that provides a dynamic container. The choice of `vector` is based on the need for an ordered container, since the AFS set is continually traversed and edited. Since a specific order is required and the edit index is known, `vector` offers greater efficiency and speed in this context, avoiding potential memory conflicts.

In the approach proposed by [8], an additional EFS set is used to store exhausted sources, along with an RP set to record pixels that were analyzed and discarded as non-edges. The main purpose of these sets is to keep track of the analyzed positions in the image, thereby preventing repeated inspection of the same locations. Instead, we choose to mark these positions directly in the output image by assigning a value of 1 to pixels that are not edges and a value of 255 to detected edges. This allows the algorithm to quickly verify if a pixel has already been analyzed by checking if its value in the output image differs from 1 and 255. The decision to mark pixels directly stems from the observation that the main factor slowing down the algorithm is the overhead associated with insertion and lookup operations in the sets. By implementing this strategy, processing time is significantly reduced, improving overall efficiency.

In the process of searching for new food sources, the use of SIMD instructions is also added, performing convolution in the same way as in the function ParallelSSEConvolution(), which accelerates detection.

To efficiently keep track of the pending neighbors to be explored for the food sources, instead of directly storing the positions in the image of each neighbor (which would lead to excessive memory usage), an array ($array < int, 9 > neighbors\_by\_chosen$) is used to hold the index corresponding to their relative position with respect to the source "fk". This index varies between 0 and 8, where 0 represents the first neighbor and 8 the last. The last position of the array is reserved to store the number of neighbors that have not yet been explored, allowing for quick identification of whether the source is exhausted. Furthermore, this structure optimizes the process of randomly selecting neighbors. To select a neighbor, a random number is generated between 0 and $neighbors\_by\_chosen[8]$ (the number of pending neighbors), and the selected index is moved to the second-to-last position of the array. In this way, the first elements of the array correspond only to the neighbors that have not been explored yet, and the count of pending neighbors is reduced by 1, as shown in Figure 3.

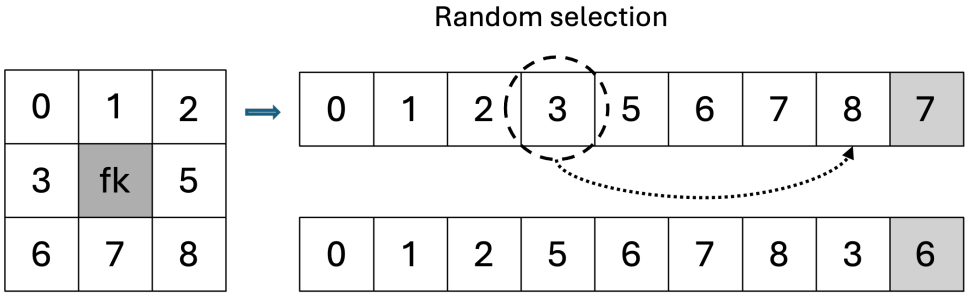

Figure 3: Random selection process of a neighbor for the source fk.

Then, based on the index of the selected neighbor, its position in the image can be accessed using Equation 5 for the horizontal position and Equation 6 for the vertical position, respectively. This could be explored in more detail in the Algorithm 6 of the GetNeighborCandidate function.

$$nfk_x = index\%3 - 1 + fk.x. \tag{5}$$

$$nfk_y = index/3 - 1 + fk.y. \tag{6}$$

Where nfk_x and nfk_y are the horizontal and vertical positions of the selected neighbor.

Algorithm 6 Candidate Neighbor Selection.

```
Input:
    IFS → Inactive Food Sources
    fk → Current food source
    u → Threshold
    image → Input image
    output_image → Binary output image
    count_inspections → Inspected pixel counter
Output:
    nfk → Candidate neighbor food source
    Updated output_image and count_inspections
Process: Neighbor Generation and Fitness Evaluation
    nfk ← New Source()
    if fk.neighbors_by_chosen[8] > 0 then
        id_vecino ← rand() mod fk.neighbors_by_chosen[8]
        offsetX ← (fk.neighbors_by_chosen[id_vecino] mod 3) - 1
        offsetY ← (fk.neighbors_by_chosen[id_vecino] div 3) - 1
        nfk.fk ← Point(fk.fk.x + offsetX, fk.fk.y + offsetY)
        Rotate(fk.neighbors_by_chosen, id_vecino)
        fk.neighbors_by_chosen[8] ← fk.neighbors_by_chosen[8] - 1
        if output_image[nfk.fk.y, nfk.fk.x] == 0 then
            count_inspections ← count_inspections + 1
            nfk.fit ← convolution3x3SIMD(image, nfk.fk.x, nfk.fk.y)
            if nfk.fit < u then
                output_image[nfk.fk.y, nfk.fk.x] ← 1
                nfk.fit ← 0
            else
                output_image[nfk.fk.y, nfk.fk.x] ← 255
            end if
        else if output_image[nfk.fk.y, nfk.fk.x] == 255 then
            source ← IFSBelongs(nfk.fk, IFS)
            if source.fk ≠ Point(0,0) then
                nfk.fit ← source.fit
                nfk.neighbors_by_chosen ← source.neighbors_by_chosen
            end if
        end if
    end if
    return nfk
end process
```

In the Scout Bees phase, it checks for potential sources to create, meaning that if the number of analyzed pixels is less than the total number of pixels in the image, the algorithm will stop, setting $SN = 0$, as it is considered that the image has been fully explored. The count of analyzed pixels ("*count_inspections*") corresponds to the number of sources found in addition to the number of pixels discarded for not being edges. To keep track of this, a counter is created that increments with each new inspection.

5 Results

In the implementation of the algorithms, the C++ programming language has been chosen due to its outstanding efficiency. The compilation is carried out using Microsoft Visual C++ 2019, integrated into the Visual Studio 2022 development environment. The device used for executing the algorithms is equipped with an Intel I5-12450H processor, 16 GB of RAM, and a frequency of 2.00 GHz.

The input image selected for testing is the grayscale Lenna image, with dimensions of 512×512 pixels. This selection criterion facilitates the comparison of the results obtained with existing methods in the state of the art.

The threshold set for the tests is fixed at 175; it is important to note that this value may vary depending on the characteristics of the analyzed images.

To accurately measure execution time, the `std::chrono` library from the C++ standard is employed, which provides specialized utilities for time measurement and manipulation. This approach ensures the acquisition of precise and reliable timing results during the evaluation of the implemented algorithms. Additionally, to avoid variations due to system load, the execution of the algorithms is repeated 50 times, and the average execution times obtained are reported.

5.1 SIMD Instructions

Edge detection using SIMD instructions is achieved effectively, as shown in Figure 4, with an execution time of 0.551 milliseconds.

(a) (b)

Figure 4: Edge detection with SIMD Instructions. (a) Original image. (b) Edges detected with SIMD.

5.2 ABC-ED

The evaluation of the ABC-ED algorithm is carried out by considering two fundamental quality measures: efficiency, which reflects the fidelity of the obtained result compared to the actual result, and the number of analyzed pixels (PA), which indicates the percentage of the image examined to achieve the attained efficiency; that is:

$$\text{Efficiency } \% = \frac{\text{Number of edges detected by ABC-ED}}{\text{Number of edges detected by the classical method}}. \tag{7}$$

$$\text{PA } \% = \frac{\text{Number of pixels analyzed ABC-ED}}{\text{Total number of pixels in the image}}. \tag{8}$$

Various tests were conducted by varying the maximum number of cycles (MCN). The obtained results, presented in Figure 5, show that starting from 80%, the model effectively identifies the representative edge structure of the image when this parameter is varied from 62 to 111. This suggests that analyzing just 20% of the image (see Table 2) is sufficient for the model to effectively represent the edges, achieving this result in a time of 13.686 milliseconds. Additionally, it is observed that as efficiency is increased, the execution time experiences a considerable rise.

Based on the innovation of this method in edge representation by analyzing the minimum number of pixels possible, it is concluded that it is not advisable to use the method above 95% efficiency, since although more precise representations are obtained, the increase in execution time does not justify the improvement in efficiency.

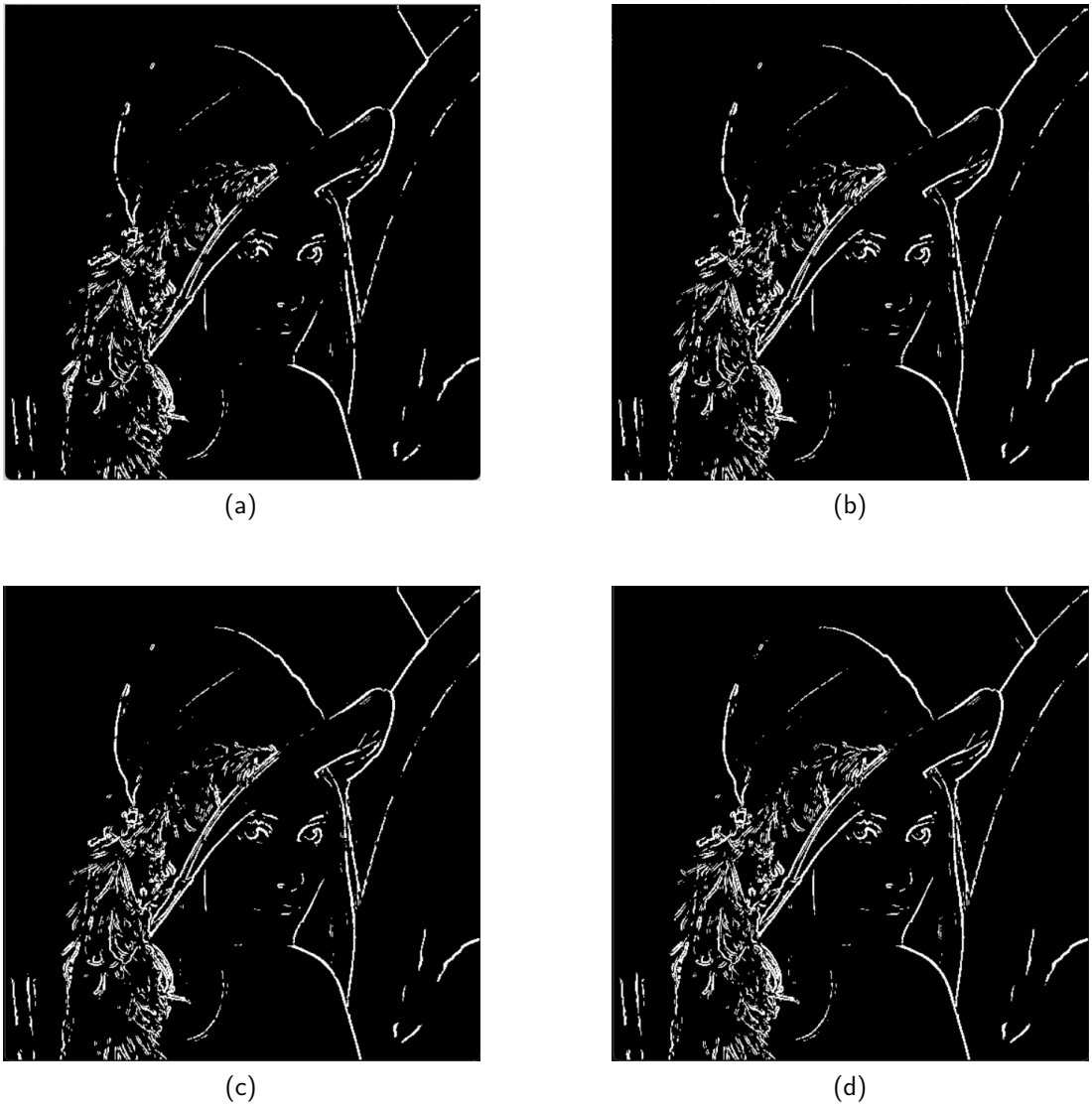

Figure 5: Edge detection obtained with the ABC-ED algorithm. (a) For 80 % efficiency. (b) 85 %. (c) 90 %. (d) 95 %.

Table 2: Results of the optimized ABC-ED algorithm.

Cycles	Efficiency(%)	PA(%)	Time(ms)
62	80	20	13.686
70	85	27	16.045
77	90	33	18.240
86	95	49	21.871
111	100	99	74.361

According to the results shown in Table 2, it can be analyzed that the execution time of the optimized ABC-ED method, despite evaluating only 20% of the image, is still high compared to the SIMD instruction method. This is due to the algorithm constantly performing operations to save, edit, and delete in different datasets, which adds execution time to the process.

5.3 Comparison with the State of the Art

The evaluation of the algorithms includes comparisons with other equivalent methods dedicated to the same process, as detailed in references [8, 10]. Furthermore, edge detection was implemented using a custom build of OpenCV 4.8.0, specifically designed to enable AVX2 optimizations. This provides greater control over instruction set utilization

and allows for further performance improvements in the evaluation. Sobel-based edge detection was implemented following the procedures outlined in the official OpenCV documentation.

The results obtained are presented in Table 3, where it is highlighted that our ABC-ED method surpasses the execution time proposed by [8]. However, the method with the best performance is our SIMD approach, which even outperforms the execution time achieved by the OpenCV library. This finding underscores the effectiveness and superior performance of our approach in relation to existing reference methods.

Table 3: Comparison of the Algorithms with the State of the Art.

Method	Efficiency (%)	Time (ms)	Dimension
ABC-ED [8]	80	965.000	512×512
ABC-ED proposed	80	13.686	512×512
SIMD [10]	100	5.299	286×384
OpenCV [17]	100	1.172	512×512
SIMD proposed	100	0.551	512×512

6 Conclusions

The results obtained from the evaluation of the proposed algorithms reveal an outstanding performance in edge detection in images. The implementation of the SIMD method has proven to be particularly efficient, even surpassing the execution time achieved by the renowned OpenCV library. This approach, by processing multiple pixels simultaneously and optimizing processor resource usage, presents an effective and fast solution for edge identification.

On the other hand, the ABC-ED method, while achieving an effective representation of edges by evaluating only 20% of the pixels in the image, must perform multiple operations on different datasets, increasing its execution time. However, its potential is recognized, and its application in more computationally complex tasks is contemplated, where the time added by the algorithm's operations is offset by the time required to perform that task over the entire search area. This acknowledgment underscores the versatility of the ABC-ED method, suggesting its consideration in specific contexts that demand a more selective approach and where its strengths can be fully leveraged.

Together, these findings highlight the importance of algorithm optimization for image processing and emphasize the effectiveness and superiority of the SIMD implementation compared to existing methods. These results not only have significant implications in the field of edge detection but also offer valuable insights for the continuous design and improvement of algorithms aimed at similar real-time tasks.

A promising direction for future work is extending the current SIMD-based implementation of SSE to more advanced instruction sets such as AVX2 and AVX-512. These architectures offer larger vector registers and higher computational performance, which could further accelerate the Sobel operator and other edge-detection processes under the same experimental conditions.

7 Data Availability

All datasets and code are available at: https://github.com/elimm1910/OptimizationMethodsEdgeDetect.git

References

[1] S. Gupta, C. Gupta, and S.K. Chakarvarti. "Image edge detection: a review". In: *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)* 2.7 (2013), pp. 2246–2251.

[2] V.M. Dharampal. "Methods of image edge detection: A review". In: *J. Electr. Electron. Syst* 4.2 (2015), pp. 2332–0796.

[3] C. Orhei et al. "Dilated filters for edge-detection algorithms". In: *Applied Sciences* 11.22 (2021), p. 10716.

[4] A. Fuentes. "Optimización de algoritmos científicos en sistemas heterogéneos y aceleradores para computación de altas prestaciones". PhD thesis. Universidad de Córdoba (ESP), 2023.

[5] P. Sabouri and H. GholamHosseini. "Lesion border detection using deep learning". In: *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2016, pp. 1416–1421.

[6] A. Banharnsakun. "Artificial bee colony algorithm for enhancing image edge detection". In: *Evolving Systems* 10 (2019), pp. 679–687.

[7] S. Kumar et al. "Optimization Methods for Image Edge Detection Using Ant and Bee Colony Techniques". In: *Advances in Information Communication Technology and Computing: Proceedings of AICTC 2022*. Springer, 2023, pp. 381–388.

[8] J. Vásquez Feijóo. "Localización eficiente en detección de bordes en imágenes adaptando el algoritmo ABC". PhD thesis. Universidad de Concepción., 2016.

[9] J. Vásquez F., R. Contreras A., and M. A. Pinninghoff J. "Efficient Localization in Edge Detection by Adapting Articial Bee Colony (ABC) Algorithm". In: *Natural and Artificial Computation for Biomedicine and Neuroscience*. Ed. by José Manuel Ferrández Vicente et al. Cham: Springer International Publishing, 2017, pp. 107–114. ISBN: 978-3-319-59740-9.

[10] M. Qi, G. Sun, and G. Chen. "Parallel and SIMD Optimization of Image Feature Extraction". In: *Procedia Computer Science* 4 (2011). Proceedings of the International Conference on Computational Science, ICCS 2011, pp. 489–498. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2011.04.051. URL: https://www.sciencedirect.com/science/article/pii/S1877050911001098.

[11] C. Mala and M. Sridevi. "Parallel algorithms for Edge detection in an Image". In: *2014 17th International Conference on Network-Based Information Systems*. IEEE. 2014, pp. 23–30.

[12] T. Peng-o and P. Chaikan. "Optimization of Edge Detection using AVX Intrinsics on Multi-core Architectures". In: *2022 37th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC)*. 2022, pp. 364–367. DOI: 10.1109/ITC-CSCC55581.2022.9894947.

[13] D. Kusswurm. "Modern Parallel Programming with C++ and Assembly". In: *Apress: Berkeley, CA, USA* (2022).

[14] D. Kusswurm. "X86-64 SIMD Programming". In: *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX*. Berkeley, CA: Apress, 2014, pp. 563–622. ISBN: 978-1-4842-0064-3. DOI: 10.1007/978-1-4842-0064-3_20. URL: https://doi.org/10.1007/978-1-4842-0064-3_20.

[15] R. Jošth et al. "Real-time PCA calculation for spectral imaging (using SIMD and GP-GPU)". In: *Journal of real-time image processing* 7 (2012), pp. 95–103.

[16] E. Welch et al. "A study of the use of SIMD instructions for two image processing algorithms". In: *2012 Western New York Image Processing Workshop*. IEEE. 2012, pp. 21–24.

[17] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).