ENERGY AND STORAGE OPTIMIZATION IN PRECISION LIVESTOCK FARMING

Julián Ramírez
Universidad Eafit
Colombia
jdramirezl@eafit.edu.co

Andrés Salazar
Universidad Eafit
Colombia
asalaza5@eafit.edu.co

Simón Marín
Universidad Eafit
Colombia
smaring1@eafit.edu.co

Mauricio Toro
Universidad Eafit
Colombia
mtorobe@eafit.edu.co

ABSTRACT

This work introduces some of the most widely used compression algorithms, and their relevance to the field of livestock farming, which has been historically characterized for requiring menial and inefficient labor, introducing environmental. And also for lacking the scale and automation that cutting edge technologies can provide. By doing this we will explain how this opens the door to locations untouched by technology, and the general advantages, and possibilities that integrating pattern recognition models bring to the table. In addition, we will explain the ins and outs of these compression algorithms, and our reasoning behind our decision to choose an algorithm to implement in our pattern recognition model.

To solve this problem, Seam Carving, Image Scaling and Run-Length encoding were used. With them we compressed the images an average of 17.5% of their original size in a time complexity of O(L*N*M). This research shows how you can create an efficient compression algorithm for usage in PLF.

Keywords
Compression algorithms, machine learning, deep learning, precision livestock farming, animal health.

1. INTRODUCTION

Year after year concerns related to the food supply arise as conventional methods of farming become insufficient with farms populated by thousands of animals. Our current reliance on antibiotics, and inefficient manual detection methods are a time ticking bomb, that unless they are replaced by modern automated methods, will result in thousands of sick animals, consequently, undermining the farming industry.

A promising solution comes with PLF. Which allows farmers to have a constant and automated method of assuring the wellbeing of all their animals even if they are in the thousands.

1.1. Problem

With the increasing concerns in population growth and climate change. Optimization comes into mind as a crucial driver to manage our resources. As previously stated, this paper will be centered around the classification of livestock. Primarily, by implementing an image recognition model that will determine whether an animal (in this case a cow) is sick or not.

Initially, we will introduce various compression algorithms, their implementations, time and space complexities. In this way we intend to display a general view of some of the most famous compression algorithms. And towards the end this will allow us to define which algorithms will be chosen, and the reason behind our decision.

The purpose behind implementing compression algorithms in this paper lies in the fact that network infrastructure in rural areas is often limited. Thus, it is imperative to send only relevant information to both query, and feed the image recognition model.

This in turn will help to distinguish healthy from unhealthy animals. Preventing the loss of cattle, and consequently aiding the ever-growing demand for meat which is one of the most polluting industries.

1.2 Solution

In this work, we used a convolutional neural network to classify animal health, in cattle, in the context of precision livestock farming (PLF). A common problem in PLF is that networking infrastructure is very limited, thus data compression is required.

As to help with energy and storage optimization we're going to implement compression algorithms. We implemented Seam Carving, Image Scaling and Run-Length Encoding.

Seam Carving, because it removes parts of the image with low energy, as in parts that are not very important such as backgrounds, floors, or parts that are generally not the cow.

Image Scaling, as it can compress big images into small ones, making the files easier to process.

And Run Length encoding because it's a computationally cheap algorithm that allows people with not very fast computers to compress these images easily.

1.3 Article structure

In what follows, in Section 2, we present related work to the problem. Later, in Section 3, we present the data sets and methods used in this research. In Section 4, we present the algorithm design. After, in Section 5, we present the results. Finally, in Section 6, we discuss the results, and we propose some future work directions.

2. RELATED WORK

3.1 Dairy Cattle Subclinical Uterine Disease Diagnosis Using Pattern Recognition and Image Processing Techniques:

In this paper, the center of attention is the prevention, and right classification of the postpartum uterine disease named endometritis. Which is defined as a chronic inflammation of the endometrium, limiting the fertility of cattle after giving birth without displaying any easy distinguishable symptoms in the afflicted cow. The conventional way they mention this is diagnosed is by using ultrasonic images of the uterus layers which can then be analyzed by a professional veterinary. Nonetheless, they propose it is possible to build an image recognition model that could automate the diagnosis of this illness by examining the appearance of the uterus layers (myometrium and endometrium), and the presence of uterine fluid (lumen) [18]. The proposed model is trained in a supervised fashion, by having an expert classify healthy from unhealthy cattle using shapes, pixel intensity, textures, among other characteristics of the images.

They mention subclinical endometritis is often characterized by the thickness of the layers of the uterus. Thus, they implemented an algorithm which describes the normal evolution of the layers, that in summary will determine the thickness of every one of the concerning regions.

Additionally, they mention that because the proportion of unhealthy cows to healthy is heavily skewed over the latter. The model could be unbalanced, heavily favoring healthy population. The way they proposed to solve this is by implementing a weighted-cost learning technique which aims to assign a higher impact when a misclassification is done by the model.

Regarding their results, they implemented two datasets. For one of them they implemented SMOTE technique (a technique used to oversample unbalanced classes) yielding 0.86 accuracy. For the same dataset without SMOTE accuracy was at 0.71. Nevertheless, for their second dataset accuracy was up at 0.98 [18].

3.2 Early detection of morbidity in feedlot cattle using pattern recognition techniques:

This paper introduces Bovine Respiratory Diseases as a significant concern regarding livestock. It mentions how newly arrived calves account for the majority of disease control issues. And how in comparison to older cattle, they have a 30 – 50 % chance of suffering from these illnesses.

In comparison, only below 30% of older animals are diagnosed to suffer from respiratory diseases. Nonetheless, they emphasize the inefficiencies of the current procedures to diagnose an animal, and how the pervasive use of antibiotics is detrimental to our supply system while not yielding acceptable results. Whereas, technologies like pattern recognition, could completely automate the process of diagnosis, and more precisely determine when the use of antibiotics is required.

This in turn would also improve the efficacy of these medicines as early detection is considered a key factor in recovery from respiratory diseases. Additionally, they stress how 40% of deaths in livestock are linked to respiratory illnesses, and how implementing pattern recognition would highly decrease those numbers.

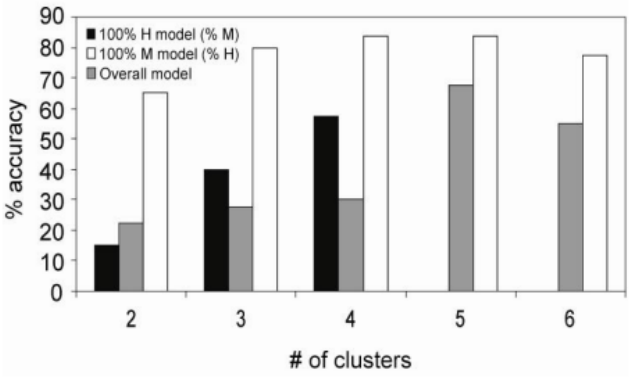
According to this paper, erratical feeding behavior is often seen in unhealthy cattle. Sick animals don't await food delivery, and don't react to its arrival. Whereas healthy ones do this, and are also found feeding upon arrival.

They propose the implementation of proximity sensors, installed in the animal's pens. That over the course of the day could accurately track the feeding patterns.

While tracking the data is important, a model to interpret it is crucial if automation is really desired. In this case, they proposed an unsupervised learning model, which creates clusters of animals according to their corresponding behaviors.

In this study, animals were classified in three groups: high, low, and medium risk groups. And for each of these the clustering algorithm was applied.

Results of the study are shown below.



[17]

3.3 Early detection of health and welfare compromises through automated detection of behavioral changes in pigs

In this document, the researchers try to research how to detect the welfare of pigs using technology, specifically sensors.

The document starts by explaining the different illness/problems a pig can have and how it transforms to behavioral changes. For example, how a pig with salmonella will spend more time sitting or standing than it normally would do, how pigs cluster together when the temperature is low, or how changes in normal vocalization could show feed deprivation, etc.

The proposed method to solve this is through different kinds of sensors located throughout the pen. These sensors would take different data from the pigs, and it could be processed through different processes to find any abnormalities.

1. Audio:

The first proposed sensor is a Microphone, which would take the sound of the pigs and through processes detect and classify specific acoustic events, such as sneezing, screaming, etc. Putting more microphones will help with the localization of sound sources. And with "spectral analysis" separating pig coughs from the rest of the audio is possible.

And its even possible to detect different types of coughs that are linked with different diseases.

2. Visual:

Using cameras, we can record and detect pig behavior. For example, the difference between pixels in different images can show if a pig is being aggressive (With fast pixel changes). But not always moving fast is aggressive, it could also only be the pig playing around, so to detect heat-to-head knocking the document proposes the use of a 3D camera.

The biggest challenge that comes with the use of cameras is the hostile environment that is a pig pen. The cameras can be damaged through the dust or ammonia generated in there but it can be countered through maintenance.

In the end, the document states that automation using sensors has the potential to detect deviations in the welfare and behavior of pigs, but that right now the approaches that monitor behavior are not advanced enough and require a person to be the judge of it [2].

3.4 Precision livestock farming in the context of meat safety assurance systems.

To make sure the "Farm-to-Fork" chain of food is safe for the consumer it is imperative to make sure the meat and the cows are healthy before being slaughtered, so this document talks about how we can connect PLF with this kind of farming to monitor the wellbeing and behavior of cows, to make sure they're healthy as a healthy animal is the first step of a safe chain of food.

The document starts by stating that PLF can detect disease early, through symptoms that the cows show. Diseases that can later be treated earlier than if it wasn't detected, and also avoiding the use of antimicrobial. PLF uses different methods for taking information, e.g., biosensors, that then measure different variables on the animals such as good health, behavior, etc.

The document later says that "is predicted the worldwide increase of animal products will be around 70% by 2050" [3] so when in the next decades a farm consists of thousands of cows and other animals, how would a group of persons really assure the wellbeing of all of them?

That is where the PLF comes in, taking real time information from any range of animals, making sure of the wellbeing of the animals. Many examples of PLF applied with cows are:

- Detecting lameness (When an animal has leg or foot pain that affects how they move) in an animal through the use of acceleration data from ear tags.

- Detecting changes in vocalization through microphones, to check deviations.

- Detecting body temperature through infrared technology

- A very important one is the early detection of pathogens, which can be done through the use of biosensors that recognizes a target biomarker for a specific pathogen. The biochemical interaction between the bioreceptor and the biomarker sends is transformed into a signal to be digitally processed.

The document concludes saying that PLF lets farmers check and make sure of the wellbeing of their animals constantly and automatically which helps them make decisions based on the real and valid data acquired by the sensors. PLF is a powerful tool which helps with the public health and the economic repercussions which can bring infected or bad meat from unhealthy animals [3].

3. MATERIALS AND METHODS

In this section, we explain how the data was collected and processed and, after, different image-compression algorithm alternatives to solve improve animal-health classification.

3.1 Data Collection and Processing

We collected data from Google Images and Bing Images divided into two groups: healthy cattle and sick cattle. For healthy cattle, the search string was "cow". For sick cattle, the search string was "cow + sick".

In the next step, both groups of images were transformed into grayscale using Python OpenCV, and they were transformed into Comma Separated Values (CSV) files. It was found out that the datasets were balanced.

The dataset was divided into 70% for training and 30% for testing. Datasets are available at

https://github.com/mauriciotoro/ST0245-Eafit/tree/master/proyecto/datasets.

Finally, using the training data set, we trained a convolutional neural network for binary image-classification using Google Teachable Machine available at https://teachablemachine.withgoogle.com/train/image.

3.2 Lossy Image-compression alternatives

In what follows, we present different algorithms used to compress images.

3.2.1 Seam carving:

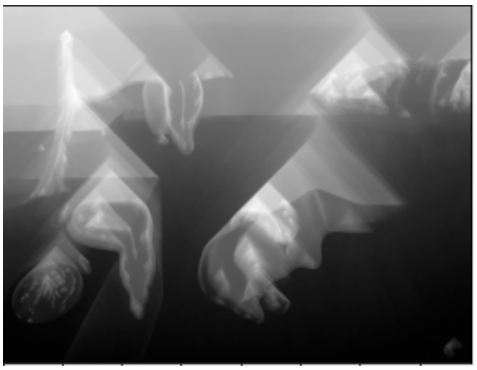
Seam carving algorithm is a compression algorithm known to be "content aware". In contrast to other techniques like resizing (which preserve objects at the cost of distorting them) or cropping (which maintains shape at the cost of cutting parts of the image), this algorithm strives to achieve the benefits of both without having the shortcomings of either.

The way in which this algorithm manages to do this can be explained in 4 simple steps:

1. Start by drawing an energy map of the image: This is done by adding the absolute value of the gradient in the x direction with the gradient of the y direction of each pixel. Gradients can be understood as significant changes in color between a pixel and their neighboring pixels. And they are calculated using a matrix called a sobel filter. The sum of the values of each pixel multiplied by the values in the matrix yields the derivative of the change in color (also known as the gradient).

Original Image

Energy map of the Image



2. Identify the "seam" in the image matrix: By using dynamic programming, this algorithm finds the path with the minimum sum of gradients among its pixels. It does so by starting from bottom to top of the energy map, and for each pixel in each row, it calculates the cumulative gradient of a pixel by adding the value of the pixel gradient with the minimum value of its three neighbors' bottom pixels.

   Once the top row is reached, the pixel with the minimum gradient value denotes the start of the seam.

   Image with seam displayed as a white line on its right side.

3. Remove pixels that belong to the seam: At this point, pixels conforming the seam represent locations in the image where empty space is more likely to occur. Thus, they can be erased from the image as they do not belong to objects while also taking up space. This can be achieved by starting with the pixel with the minimum gradient at the bottom, deleting it, then moving to its neighboring pixel to the bottom with the minimum value, and repeating until the bottom row is reached.
4. Repeat for the required number of seams.

The time complexity of this algorithm is O(H*W), H being the height of the image, and W being the width of the image. This is the case because in general terms, it is necessary to traverse the entire image four times. One to calculate the x gradients, another to calculate the y gradients. A third one to draw the energy map, and a final to remove the seam from the image. Note, however, that the number of operations in each traversal remains constant, and for that reason the overall time complexity remains at O(H*W).

Sample Image:

In terms of space complexity, since an energy map is required. It turned out to also be O(H*W) additional space.

Shortcomings of this algorithm:

1. It only removes pixels in the x-axis. If vertical compression is desired, it may be required to rotate the image.
2. Using this algorithm for images with high contrast or poor definition of objects, result in distortions to some shapes in the image.

[8,8,19,25,26]

3.2.2 Image Scaling

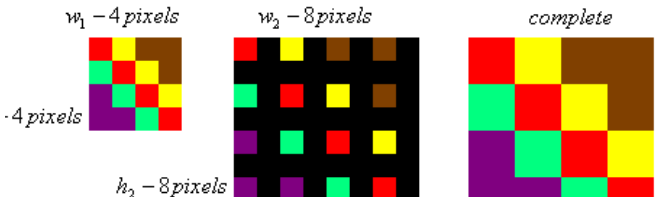
Please explain the algorithm, its complexity and include a vector Figure.

Image Scaling algorithm is a procedure by which an image's dimensions can be refactored to a scale in the intervals of (0, ∞). When the scaling ratio is between the values 0 and 1, the algorithm behaves as a compression technique. On the other hand, when this ratio is above 1, the algorithm acts as a decompression technique.

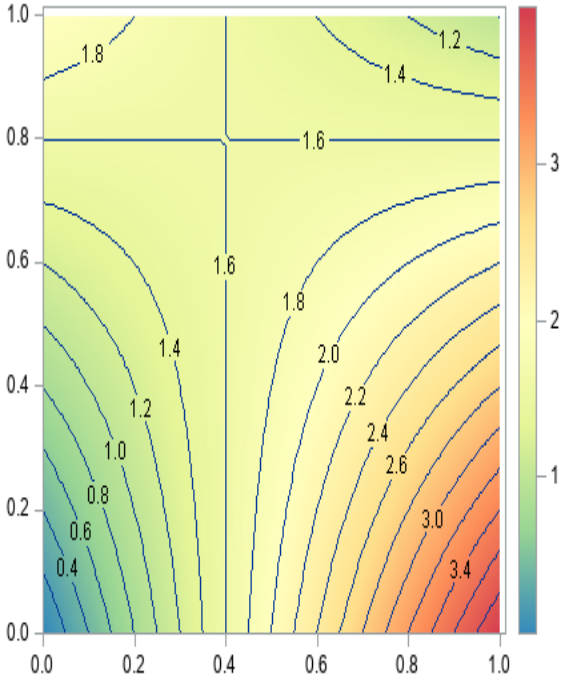
The essence of this algorithm lies in the fact that an image can be resized by multiplying the positions of each pixel by the scaling ratio. Then setting those positions in the new image to the value in the original position of the original image. And finally, filling the gaps using one of the methods to approximate each blank pixel to the color of its closest pixel.

There are three commonly used procedures to approximate a blank pixel:

1. Nearest neighbor: Set the value of the blank pixel to the value of its closest nonblank neighbor. This technique is very straightforward. However, it often results in pixilation of the output image.



2. Bilinear interpolation: Assign to the value of the blank pixel, the coordinate of the function created by the difference in values of its two closest pixels divided by the number of pixels among them. (i.e., if a pixel p3 lies on the third position between two pixels p1, and p2, with their respective values 200, and 220. And the distance between said pixels is 10 pixels, the value of p3 would be described as follows: ((220 – 200)/10) * 3). This method is effective in decreasing pixilation. Nonetheless, when using big scaling factors, a shortcoming that this method suffers is that sharp changes in color can be detected.



3. Bicubic interpolation: Based on the tangents of the values of a pixel with its neighbors (gradient), and the value of the pixel itself. A transformation of the discrete values of the pixels, to an approximation of a cubic graph in the pixels between two non-blank pixels (also referred to as generating a cubic spline) is produced. This in turn preserves the benefits of a bilinear interpolation, while also preventing abrupt changes in pixel

values. Nevertheless, this interpolation method also comes with its shortcomings. One of them is that by inserting valleys and peaks of a curve into the output image. Information that is non-existent in the original image is introduced to the output image.

It is important to highlight that these operations are required for compression and decompression procedures because for both of their output matrices, blank pixels are generated. However, they are far less necessary for compressions, as it generally emphasizes the deletion of pixels.

Depending on the interpolation method, time and space resources will vary. Nearest neighbor and Bilinear interpolation being the most efficient ones. Nonetheless, it is safe to assume that any of them stay in the realm of O (W * H) for both time and space complexity. With some additional factors for bicubic method as it is said to require computations of matrices to generate the interpolated pixels.

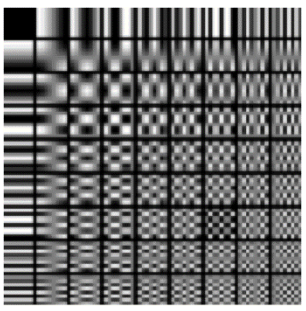
[6,15,27–29]

3.2.3 Discrete Cosine Transform

Direct Cosine Transform algorithm (DCT) argues that any number of discrete points in a graph can be represented as the approximation of the weighted sum of one or more cosine functions at different frequencies. And that higher frequencies added to the overall function have a lower impact in the outcome. Thus, making it possible to completely remove them from the equation without having a significant visible change.

In a 2D plane, this is carried out by combining the cosine functions for both the x-axis and the y-axis at different frequencies. The standard to this is to generate all the permutations in an 8 x 8 matrix, also generally called the DCT matrix. Each block in this matrix also contains an 8 x 8 matrix that will have multiple purposes further in the algorithm.

Sample DCT matrix.



Regarding the image to be compressed, this algorithm follows the following steps:

1. Divide the entire image in 8 x 8 blocks of pixels: As discussed before, each block in the DCT matrix is also composed of 8 x 8 blocks. Thus, it is imperative to work with an image that can be divided in this size of blocks because most operations require a one-to-one relationship between the cells of the current block and the cells of a given block of the DCT matrix. Some algorithms work around this problem by appending pixels at the end of the rows and columns, therefore allowing any type of image to be used.

2. Center all values in each block: RGB colors range from 0 to 255 values for each type of color (green, blue, red). In order to be compatible with the values given in the DCT matrix, it is necessary to deduct 128 from each value. This is the case because the cosine function is centered around 0. Moreover, all the values in the matrix of coefficients will be centered around 0.

3. For each 8 x 8 block in the image apply the DCTII function: As previously mentioned, this algorithm argues that any 8 x 8 matrix of pixels can be represented as the sum of one or more blocks in the DCT matrix. The DCTII algorithm finds the weights assigned to every block in the matrix of coefficients. This in turn will create a new matrix which we will call the matrix of coefficients. Each value in this matrix multiplied by its corresponding block in the DCT matrix yields the block from which it was created in the original image. The DCTII algorithm creates the matrix of coefficients by traversing each 8 x 8 block from left to right, top to bottom and applying the formula shown below.

$$T_{ij} = \begin{cases} \frac{1}{\sqrt{N}} & \text{if } i = 0 \\ \sqrt{\frac{2}{N}} \cos\left[\frac{(2j+1)i\pi}{2N}\right] & \text{if } i > 0 \end{cases}$$

Int this formula, variables i and j represent the rows and columns in the DCT matrix. The resulting matrix is then multiplied by the block of the image in the following fashion: Matrix of Coefficients = T * M * T^-1. Where T is

the resulting matrix, M is the image block and T^-1 being the transpose of the resulting matrix.

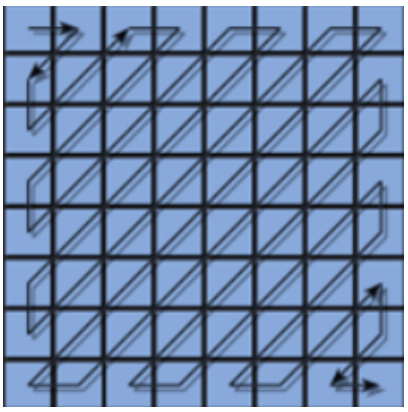
4. Apply the quantization matrix: Divide each cell in the matrix of coefficients by each element in the quantization matrix. A quantization matrix is an industry standard that will assign a value to each element in the coefficient matrix. It generally benefits cells closer to the upper-left corner of the matrix as those usually have a higher value in the matrix of coefficients. The quantization matrix will determine the resolution of the final block as it will reassign weights in the matrix of coefficients.

5. Round down each number in the matrix of coefficients: In order to disregard cosine functions with higher frequencies, the act of rounding down the values in the DCTII matrix will further reduce their importance in the decompressed image. Some elements, especially those close to the bottom-right corner of the matrix, may be rounded down to 0 altogether. Therefore, making their contribution null in the decompressed image.

This step is where the algorithm becomes lossy, as rounding down the contributions of certain cells will irreversibly change the decompressed image. Nevertheless, it is important to note that these changes are miniscule, and in most cases cannot be perceived by a human.

6. Compress each 8 x 8 matrix of coefficients: Apply other compression techniques like the Huffman coding algorithm to further compress the information. Essentially, this is achieved by converting the matrix to an array following the pattern shown below. It is done in this way, as it can be argued that adjacent blocks will have similar, if not equal values, allowing for a more efficient compression process.

Compression pattern

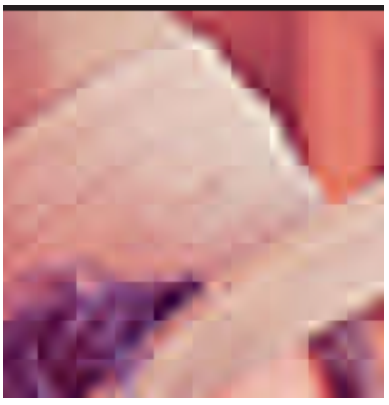


The decompression process follows a reversed process of the one depicted here. And the accuracy of the output image will be dependent on the precision of the quantization matrix used.

Sample images decompressed using this algorithm:

Low precision quantization matrix -

Medium precision quantization matrix



High precision quantization matrix

The classical 2D discrete cosine transform algorithm follows a time complexity of O (N^2 log2(N))[14]

[5,9,14,30]

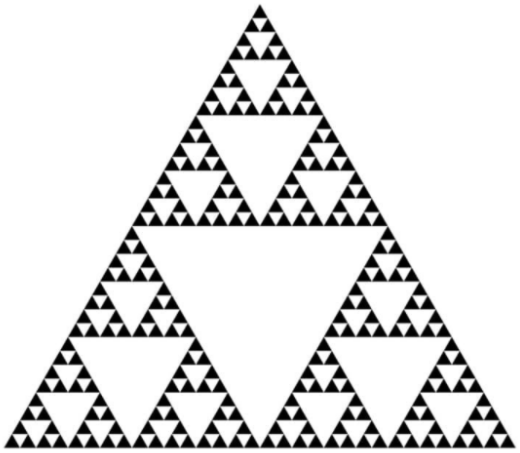
3.2.4 Fractal compression

Please explain the algorithm, its complexity and include a vector Figure.

Before diving into the complexities of this algorithm, it is important to understand the concept of a fractal.

A fractal is a geometric structure that can be zoomed indefinitely without losing its shape

Example Sierpinski Fractal



In nature, fractals do exist. However, the extent to which they can be zoomed is limited, as being infinitely zoomable would involve having infinite resolution (which obviously doesn't occur).

Now, the way in which this algorithm implements fractals, is by iterating through blocks of the picture (also called domain), and looking for patterns that when decoded with an inverse function, they would yield a compressed image (range). This main function is usually called an Iterative Function System (IFS),

An important thing to notice is that this algorithm doesn't compress exact information, but shapes that approximate to a possible fractal. Hence, its presence in the lossy algorithm's category.

As mentioned above, each block iterates over itself looking for fractal patterns. For this reason, this algorithm is considered to be of O(n^2) time and O(n) space.

Some advantages of this compression method is that it does not lose resolution when scaled to bigger sizes. Because the decompression function can be implemented over and over, the output images tend to be smooth, and give great results where other algorithms may fail.

The reason behind this is that the decompression function represents the mathematical representation of parts of the image as a fractal. Which, as we mentioned before, have the property of remaining visibly equal when zoomed in or out.

Section of Image resized with fractal compression.

The Same section of the image is resized with a different compression algorithm.

[16,31,32]

3.3 Lossless Image-compression alternatives

In what follows, we present different algorithms used to compress images.

3.3.1 Run-length encoding

This algorithm is simple, it consists of replacing occurrences of a same character with the format: Frequency+Character.

Let us see an example:

AAAABCCCCCBBBAAACB

We start with a repetition of 4 A's, so we replace the repetitions with the format:

4ABCCCCCBBBAAACB

Now we've shortened the string from 4 "A"'s to one number and one "A". We keep doing this until we finish.

4A1B5C3B3A1C1B

This is our final product, a combination of numbers and letters that is generally shorter than our input string (This can fail for example with the string "ABC" because it changes to "1A1B1C", so it is bigger) and the decompression is just iterating through the output writing the number of characters shown.

The Time Complexity is O(n) as you would only need to pass one time checking every character and adding it to a new string, that is it! [20]

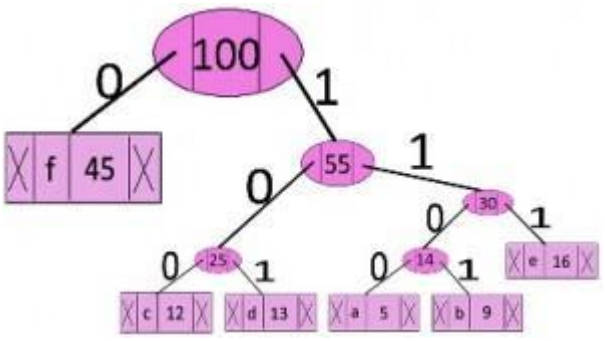
3.3.2 Huffman coding

The Huffman coding takes a set/list of symbols (Characters) and their respective probabilities/weights/Frequencies and outputs a binary tree. The process of this algorithm is that using a Priority Queue, we organize the characters according to their weight, and while there is more than one node/element in the Queue, we take the two in the top (Two with the least probabilities) and we merge them into a single element/node with a total probability of the sum of the nodes we just merged. We continue this process until we only have one Node in the Queue, which is the header node, and with the tree done we can traverse it assigning 0's to every left traversal, and 1's to every right traversal doing so until we arrived at the character we wanted, which would be a Leaf node, assigning the combination of 1's and 0's as a code word for that character. So, for example, if we had a character 'X' where the way to get to it is: Left, left, right, left its code word would be 0010.

The time complexity of the algorithm would be O(n logn) as inserting elements into the Priority Queue is LogN, and we need to do it for n nodes in the tree. If the input of characters and their probabilities is already sorted, we can use two queues, one for single characters and the other for combined characters (Nodes) and this would result in a O(n) time.

For a table like this of characters and their frequencies, this would be its Huffman Tree.

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45



To decode a compressed string, we would need the encoded string (Let us say "00101") and the tree (These two elements still weight less than the original file), and with these two we just simply traverse both the string and the

tree at the same time, taking rights at every 1 and lefts at every 0, and each time we reach a leaf node we move the pointer in the tree to the top. We could say the encoded string is an instruction to move through the tree

[7,21,22,33]

3.3.3 LZ77 and LZ78

In this section we are going to see two algorithms, the LZ77 and the LZ78.

LZ77:

The LZ77 is a sliding window algorithm which tries to do a sort of pattern-searching, checking for past occurrences of a current element.

The algorithm first divides the input into a search buffer (All the characters the window has already seen before) and a look ahead buffer (The characters it has not seen/Will check).

For every character/string it has not seen/it matched it outputs a format in the form:

(O, L, C)

Where:

- O: Offset, the number of characters/Positions the pointer would need to go back to find the matching character/string
- L: Length, the size of the match (1 if it's only a character or >1 if it's a string)
- C: Character, the character that comes after the match has ended.

For every character we need to follow the instructions above. We can better understand it with an example:

- Search buffer: []
- Look ahead buffer: {}
- Current Character: **

[] {*a* b a b c}
Output:

As our search buffer is empty (We just started) we do not have anything to look behind to match, so we output (0, 0, a) meaning: Go back 0 positions, write 0 characters from that position and afterwards write the character "a".

[a] {*b* a b}
Output:(0, 0, a)

This time the search buffer is not empty but going through it there's no 'b', so we output the same thing as the last 'a': (0, 0, b), the output explanation is the same.

[a b] {*a* b c}
Output:(0, 0, a) (0, 0, b)

This time around we indeed have an 'a' in the search buffer, which means we have a match, but after our current 'a' there is also a 'b' which is also in our search buffer. After that 'B' the search buffer ends, and we output: (2, 2, c) which means: Go back 2 positions (Where the first a is located), print 2 characters from the start, and after printing those 2 characters print a 'c':

[a b a b c] {}
Output:(0, 0, a) (0, 0, b) (2, 2, c)

With this, we have ended the compressing passing from 5 characters to 3 instructions, where we can change the characters to their ASCII values and the numbers to binary to further lower memory. For decompressing, we only need to follow the instructions in the order we outputted it.

This method has a problem that comes with very large inputs (Which is what we expect) and that is time vs space. As I mentioned above, we need to search through the search buffer to find matches, but what if there were a thousand characters already? And if the matching character was the last one in the buffer? In the end we would be doing O(k-1) for the search, so what we could do is limit both the size of the search and look ahead buffer.

a [b y] {*a* b} c

In this example if we limited the buffer to only 2 characters, we wouldn't be able to find the 'a' that is 3 positions behind and match it with the next 'b' (forming [a b]) and we would have to output (0, 0, a) (0, 0, b) and (0, 0, c) instead of (3, 2, c), 3 triplets vs 1. In the end it is a decision of time vs memory.

The time complexity of this algorithm is O(n^2) if both search and look ahead buffers are unlimited, and O(n) if they have a limited size (As the search would be a constant number of characters) .
[1,3,4,10,34,35]

LZ78:

The LZ78 comes from the idea to optimize the LZ77, as to decompress this last one we need to follow the output strictly, and we cannot start from any of the items other than the first one, not only that but as I mentioned the need to limit both the search and look ahead buffers in the compressing stage. With all these problems, Lz78 comes as a solution that does not require any order.

This approach uses a Dictionary method where every time it checks a character, it checks if there is already an entry to it in the dictionary, if it exists it keeps on adding characters until it forms a string that is not previously in the dict. The

dictionary comes normally in the form of a Hash Map and generates outputs of the form:

(pn, c)i

Where:

PN: Parent Node → Dictionary index where the longest string has been found.

C: Character → Last character that was not in the longest matching string

i: Index → Index referring to the position of this String/output in the dictionary.

We can better understand the concept with an example. We will use "{}" as a look ahead buffer, "[]" as an Already-Seen buffer and "* *" to show our actual element.

Let us start with a simple input: "ABBCBCABA".

[] {*A*BBCBCABA}
Dictionary:
Output:

As our first element "A" is not on the dictionary we output (0, A)1. First, 0, because there is no index in the dictionary where we can refer to find 'A'; Second, "A", as it means to write the characters in index 0 (Which there aren't as index 0 doesn't exist in our dictionary) and afterwards write an A. Then we put the character to the dictionary.

[A]{*B*BCBCA}
Dictionary: (1, a)
Output: (0, a)1

In our second Iteration we find that B isn't in our dictionary and the output and explanation is the same as in the first one: (0, b)2.

[AB]{*BC*BCA}
Dictionary: (1, a) (2, b)
Output: (0, a)1 (0, b)2

This time around we actually find the character "b" so we don't stop yet, and we add the next character, "c", forming "bc" which we look up and don't find in the dictionary, so we output the new string with (2, c)3, meaning: We can find the last longest string in the index 2 ("b" out of "bc"), after writing the string in the index 2 we write a c ("b"+"c") and we write the new string to the dictionary in the index 3.

[ABBC] {*BCA*}
Dictionary: (1, a) (2, b) (3, bc)
Output: (0, a)1 (0, b)2 (2, c)3

The last and final output refers to "bca", we know "b" exists, we know "bc" exists, but there is no "bca" so we output (3, a)4.

[ABBC BCA] {}
Dictionary: (1, a) (2, b)
(3, bc) (4, bca)
Output: (0, a)1 (0, b)2 (2, c)3 (3, a)4

But in the end, we do not write the actual output but instead: the binary representation of the parent index, the ASCII value of the char and finally the binary representation of the position of the string in the dictionary. (Pn, C)i changes to (binary, ASCII) binary

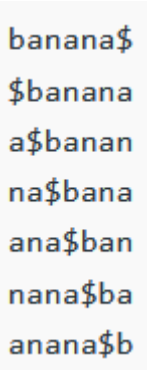
The time complexity of this algorithm is O(n), as we need to pass through N characters of the string and insert/search in an unordered map is O(1). With the map there also come collisions, so the amortized time would be O(n*k) where n is the length of the characters and K the internal map array that it would need to pass through .
[2,36–38]

3.3.4 Burrows–Wheeler transform

The Burrows Wheeler transform is not a compression algorithm itself, but a way to restructure or, as its name says, *transform* an input string into a version of itself that has sequences of the same character happen close to each other. This algorithm is only a helper to the real compression algorithm. Normally it is paired with the move to front transform and finally sent to the Huffman encoding. A very special aspect of this transform is that it is reversible if we use an EOF (End of file) character.

The way the algorithm works is (We will use the famous banana$ example):

1. Receives an input string and adds an EOF character to the end (For example purposes I will use $ but an EOF NEEDS to be unequal to any valid character), we need this EOF character for the decoding phase (To know when to stop the decoding)
2. Creates all rotations of the string:



3. Now we sort all the phrases lexicographically:

$banana		a
a$banan		n
ana$ban		n
anana$b		b
banana$		$
na$bana		a
nana$ba	------->	a

3. The last column is the BWT (Burrows-Wheel Transform)

```
BWT(banana$) = annb$aa
```


```
BWT(banana$) = annb$aa
```


To implement the method above the Time Complexity would be O(K^2) as we first need a loop that goes through the string K – 1 times (K = length of the string) to find all the rotations of the string, and in each iteration it would need to create a slice/substring (Which is O(k) itself), afterwards we need to sort the list of strings but because it is out of the loop we don't take it into the account

[10,13,23,24,39,40]

4. ALGORITHM DESIGN AND IMPLEMENTATION

In what follows, we explain the data structures and the algorithms used in this work. The implementations of the data structures and algorithms are available at GitHub[1].

4.1 Data Structures

4.2 Algorithms

In this work, we propose a compression algorithm which is a combination of a lossy image-compression algorithm and a lossless image-compression algorithm. We also explain how decompression for the proposed algorithm works. The data structures used for our algorithms are simple matrices which serve multiple purposes, going from simple ways to store data, to serving as graphs, and dynamic programming structures to optimize the performance of our algorithms.

4.2.1 Lossy image-compression algorithm

[1]https://github.com/jdramirezl/ST0245-001/tree/master/proyecto

The lossy image compression method we decided to implement was the seam carving algorithm. This method has the advantage of being object aware. Therefore, irrelevant information such as backgrounds and floors will not be included as much when pictures are sent over the internet. This method does however have two shortcomings. One of them being its time complexity, which is O(n x m x l), being n and m the dimensions of the image, and l the number of seams to be removed from the image.

Despite being an algorithm that uses dynamic programming, and caches to optimize computations. it can be slow to compress a single image. In addition to this, although irrelevant information is removed. the rates of compression remain low in comparison to other algorithms. (i.e. removing over 30% of an image may start to introduce some distortions to an image, while being computationally demanding).

For these reasons, we have to decide to incorporate the image scaling algorithm alongside seam carving. Allowing us to exploit the advantages of seam caring, while also having access to higher rates of compression.

Seam carving algorithm starts by applying a sobel filter to the image. Thus creating an energy map of the objects present in the image. It uses said map along with a dynamic programming matrix to determine the lowest energy path

from top to bottom. This path will be removed from the image at the end of the algorithm.

In the case of Image scaling, the algorithm works as follows: First multiply the i, j indexes of each pixel by a compression or decompression factor, and place the old i, j value on the coordinates of the new image. If the processing is of compression, the algorithm ends there. If it is decompression, a simple BFS algorithm is applied to fill the empty pixels around the coordinates for each pixel in the image, having a cap of layers of the resizing factor divided by two, in order to assign color values of blank pixels to their nearest neighbors.

4.2.2 Lossless image-compression algorithm

The lossless compression algorithm implemented was Run-Length encoding. We first load a CSV filled with the fray-scale values of an B&W image into a list-of-lists, and then we perform the algorithm.

The algorithm works by checking numbers, and if there is a series of repeated numbers, we replace all these numbers by writing the repeated number, and next to it the number of repetitions of it, which is written in a string with a '#' as before the number, as to differentiate between the number of repetitions and the number to be repeated. Numbers with no repetitions are written exactly as they were, with no 1 next to them. We copy these values into a list, and the list into another list, which creates a list of lists: The compressed output. Finally, we output this list of lists into a CSV and into the PC.

Again we read the CSV of the compressed output and load it to a list of lists.

To decompress we only have to iterate through these numbers, and if at the ith position what we read has a '#' as the first character, it means it is a number of repetitions and that we need to write i-th times the (i+1)-th value, any other value that doesn't have a number of repetitions written before it, it's just copied once. All these values are copied into a list of lists which is outputted to a designated output as a CSV.

```
1  254,254,254,254,254,254,254,254,250,248,255,254,247,253,205,
2  254,254,254,254,254,254,254,254,250,249,255,254,247,252,203,
3  254,254,254,254,254,254,254,254,251,250,255,254,248,250,200,
4  254,254,254,254,254,254,254,254,252,251,255,253,250,248,196,
5  254,254,254,254,254,254,254,254,254,252,255,253,251,246,192,
6  254,254,254,254,254,254,254,254,255,253,254,253,252,244,188,
7  254,254,254,254,254,254,254,254,255,253,253,252,253,242,185,
8  254,254,254,254,254,254,254,254,255,254,253,252,254,242,183,
9  254,254,254,254,254,254,254,254,250,251,250,253,255,243,181,
10 254,254,254,254,254,254,254,254,251,251,250,253,255,243,180,
11 254,254,254,254,254,254,254,254,252,251,251,253,255,241,179,
12 254,254,254,254,254,254,254,254,252,252,251,253,255,240,177,
13 254,254,254,254,254,254,254,254,253,253,251,253,255,239,176,
```

Fig. Original grayscale of image

```
1  254.0,255.0,254.0,253.0,205.0,160.0,#2,157.0,156.0,#2,155.0,154
2  254.0,249.0,254.0,252.0,203.0,160.0,155.0,#2,154.0,153.0,152.0,
3  254.0,251.0,248.0,250.0,200.0,160.0,#3,153.0,152.0,151.0,#2,150
4  254.0,251.0,250.0,248.0,196.0,160.0,#2,156.0,155.0,154.0,#2,153
5  254.0,252.0,251.0,246.0,192.0,159.0,#3,160.0,159.0,158.0,157.0,
6  254.0,253.0,252.0,244.0,188.0,159.0,#2,163.0,162.0,161.0,#2,160
7  254.0,#2,253.0,242.0,185.0,159.0,162.0,#2,161.0,160.0,159.0,#2,
8  254.0,255.0,254.0,242.0,183.0,#3,159.0,#2,158.0,157.0,156.0,155
9  254.0,251.0,255.0,243.0,181.0,#2,162.0,#2,164.0,159.0,152.0,150
10 254.0,253.0,255.0,243.0,180.0,#2,161.0,#2,162.0,157.0,152.0,150
11 254.0,253.0,255.0,241.0,179.0,#3,159.0,158.0,155.0,151.0,150.0,
12 254.0,253.0,255.0,240.0,177.0,158.0,157.0,156.0,154.0,151.0,#2,
13 254.0,253.0,255.0,239.0,176.0,155.0,154.0,152.0,149.0,#2,148.0,
```

Fig. Compressed grayscale after Seam Carving and Run Length

4.3 Complexity analysis of the algorithms

Among all the compression and decompression algorithms, Seam carving is by far the most resource intensive of them. Although it performs a linear scan of every single pixel in an image, it has to do so every time it desires to remove a seam from the image (representing L in time complexity). Additionally, in the used implementation, the numba library was implemented which increased complexity in the amortized time by an S factor. Nonetheless, it caches function calls, allowing for further optimization. In contrast, all the other implements of image scaling and run length consume O(N* M) time, as they only traverse the image matrix once to operate.

The time complexity in Image-Scaling Decompression is A*B where A = N * factor and B = M * factor, where the factor is the number of times we want to augment the size of the image

Algorithm	Time Complexity
Seam Carving	O(L*N*M) Amortized: O(S*L*N*M)
Image-Scaling Compression	O(N*M)
Run-Length Compression	O(N*M)
Run-Length Decompression	O(N*M)
Image-Scaling Decompression	O(A*B)

Table 2: Time Complexity of the image-compression and image-decompression algorithms.

Algorithm	Memory Complexity
Seam Carving	O(N*M)

Image-Scaling Compression	O(N*M)
Run-Length Compression	O(N*M)
Run-Length Decompression	O(N*M)
Image-Scaling Decompression	O(N*M)

Table 3: Memory Complexity of the image-compression and image-decompression algorithms.

In terms of memory, all the algorithms consume similar amounts of memory in their implementations (O (N*M)). Although seam carving requires an energy map, and a target matrix for every removed seam, this does not increase algorithmic complexity as constants can be dropped from the equation. Similarly, after every seam is removed, memory deallocation goes into place, allowing for further optimizations in memory. Image scaling decompression is another case where additional memory is consumed without increasing the complexity as a set is used to support the traversal of the image in the processing of filling pixels with their nearest neighbor.

4.4 Design criteria of the algorithm

The algorithms we chose were used for specific reasons.

Seam Carving, because it removes parts of the image with low energy, as in parts that are not very important such as backgrounds, floors, or parts that are generally not the cow.

Image Scaling, as it can compress big images into small ones, making the files easier to process whilst being a relatively efficient algorithm that is easy to implement.

And Run Length encoding because it's a computationally cheap algorithm that allows people with not very fast computers to compress these images easily.

In conjunction, these algorithms provide a compression rate of around 75% efficiently. Allowing for easy processing of ML algorithms.

5. RESULTS

5.1 Model evaluation

In this section, we present some metrics to evaluate the model. Accuracy is the ratio of the number of correct predictions to the total number of input samples. Precision. is the ratio of successful students identified correctly by the model to successful students identified by the model. Finally, Recall is the ratio of successful students identified correctly by the model to successful students in the data set.

5.2 Execution times

In what follows we explain the relation of the average execution time and average file size of the images in the data set, in Table 6.

Compute execution time for each image in GitHub. Report average execution time Vs average file size.

	Average execution time (s)	***Average file size (MB)***
Compression	97 s	12.4 MB
Decompression	8.9 s	12.4 MB

Table 6: Execution time of the Seam Carving, Image Compression and Run Length algorithms for different images in the data set.

5.3 Memory consumption

We present memory consumption of the compression and decompression algorithms in Table 7.

	Average memory consumption (MB)	***Average file size (MB)***
Compression	173.9 MB	3.12 MB
Decompression	29 MB	878.12 MB

Table 7: Average Memory consumption of all the images in the data set for both compression and decompression.

5.3 Compression ratio

We present the average compression ratio of the compression algorithm in Table 8.

	Healthy Cattle	***Sick Cattle***
Average compression ratio	1:4	1:4

Table 8: Rounded Average Compression Ratio of all the images of Healthy Cattle and Sick Cattle.

6. DISCUSSION OF THE RESULTS

In the end we obtained an approximately 75% compression for every image with a peak memory consumption of 173.9 MB. This is a very appropriate result as people with low end PCs will be able to run the program with little to no issues. Also, the compression ratio itself yields great results with how images are being reduced to an average of a quarter of its original size (E.G from 1 GB of images having to be sent to only 250 MB).

An important point is that results will change according to how many seams we remove in Seam Carving and how much we reduce the image size in Image Compression. Nonetheless, we consider this algorithm would perform adequately under the mentioned circumstances.

6.1 Future work

For lossy algorithms, we consider that using an implementation of the discrete cosine transform or the

Fourier transform could further optimize compression rate, and could be a great alternative to the seam carving algorithm, as they are less computationally expensive while also providing great rates of compression.

For lossless algorithms we could further improve the compression with an algorithm that does not only look for contiguous repetitions, as Run Length does, but for all repetitions or sequences in a line such as LZ77 providing further compression.

ACKNOWLEDGEMENTS

None.

REFERENCES

[1] Dhanesh Budhrani. 2019. How data compression works: exploring LZ77. *Medium*. Retrieved February 14, 2021 from https://towardsdatascience.com/how-data-compression-works-exploring-lz77-3a2c2e06c097

[2] Dhanesh Budhrani. 2021. How Data Compression Works: Exploring LZ78. *Medium*. Retrieved February 14, 2021 from https://medium.com/swlh/how-data-compression-works-exploring-lz78-e97e539138

[3] Suman M Choudhary, Anjali S Patel, and Sonal J Parmar. 2015. Study of LZ77 and LZ78 Data Compression Techniques. 4, 3 (2015), 5.

[4] Computerphile. 2013. *Elegant Compression in Text (The LZ 77 Method) - Computerphile*. Retrieved February 14, 2021 from https://www.youtube.com/watch?v=goOa3DGezUA

[5] Computerphile. 2015. *JPEG DCT, Discrete Cosine Transform (JPEG Pt2)- Computerphile*. Retrieved February 14, 2021 from https://www.youtube.com/watch?v=Q2aEzeMDHMA

[6] Computerphile. 2016. *Resizing Images - Computerphile*. Retrieved February 14, 2021 from https://www.youtube.com/watch?v=AqscP7rc8_M&feature=youtu.be

[7] CSBreakdown. 2015. *Huffman Coding - Greedy Algorithm*. Retrieved February 14, 2021 from https://www.youtube.com/watch?v=dM6us854Jk0

[8] Avik Das. 2019. Real-world dynamic programming: seam carving. *Medium*. Retrieved February 14, 2021 from https://medium.com/swlh/real-world-dynamic-programming-seam-carving-9d11c5b0bfca

[9] Dr. Ajay Kumar Verma. 2020. *Discrete Cosine Transform (DCT) of Images and Image Compression (Examples with MATLAB codes)*. Retrieved February 14, 2021 from https://www.youtube.com/watch?v=5qfUprgdTdI

[10] Google Developers. 2015. *Burrows-Wheeler Transform (Ep 4, Compressor Head) Google*. Retrieved February 14, 2021 from https://www.youtube.com/watch?v=4WRANhDiSHM

[11] Stephen G. Matthews, Amy L. Miller, James Clapp, Thomas Plötz, and Ilias Kyriazakis. 2016. Early detection of health and welfare compromises through automated detection of behavioural changes in pigs. *Vet. J.* 217, (November 2016), 43–51. DOI:https://doi.org/10.1016/j.tvjl.2016.09.005

[12] I. Nastasijevic, I. Brankovic Lazic, and Z. Petrovic. 2019. Precision livestock farming in the context of meat safety assurance system. *IOP Conf. Ser. Earth Environ. Sci.* 333, (October 2019), 012014. DOI:https://doi.org/10.1088/1755-1315/333/1/012014

[13] Mark Nelson. 1996. Data Compression with the Burrows-Wheeler Transform. *Mark Nelson*. Retrieved February 14, 2021 from https://marknelson.us/posts/1996/09/01/bwt.html

[14] Chao-Yang Pang, Zheng-Wei Zhou, and Guang-Can Guo. Quantum Discrete Cosine Transform for Image Compression. 31.

[15] Pankaj Parsania and P. Virparia. 2018. Computational Time Complexity of Image Interpolation Algorithms. *Int. J. Comput. Sci. Eng.* 6, (July 2018), 491–496. DOI:https://doi.org/10.26438/ijcse/v6i7.491496

[16] Shahid Mobin. 2017. *54. Digital Image Processing: Fractal Image Compression*. Retrieved February 14, 2021 from https://www.youtube.com/watch?v=FprMRMI0714

[17] Reka Silasi. 2007. Early detection of morbidity in feedlot cattle using pattern recognition techniques. (2007). Retrieved February 14, 2021 from https://harvest.usask.ca/handle/10388/etd-11282007-102622

[18] Matías Tailanián, Federico Lecumberry, Alicia Fernández, Giovanni Gnemmi, Ana Meikle, Isabel Pereira, and Gregory Randall. 2014. Dairy Cattle Sub-clinical Uterine Disease Diagnosis Using Pattern Recognition and Image Processing Techniques. In *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications* (Lecture Notes in Computer Science), Springer International Publishing, Cham, 690–697. DOI:https://doi.org/10.1007/978-3-319-12568-8_84

[19] The Julia Programming Language. 2020. *Seam Carving | Week 2 | 18.S191 MIT Fall 2020 | Grant Sanderson*. Retrieved February 14, 2021 from https://www.youtube.com/watch?v=rpB6zQNsbQU

[20] 2010. Run Length Encoding. *GeeksforGeeks*. Retrieved February 14, 2021 from https://www.geeksforgeeks.org/run-length-encoding/

[21] 2012. Huffman Coding | Greedy Algo-3. *GeeksforGeeks*. Retrieved February 14, 2021 from https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

[22] 2017. Huffman Decoding. *GeeksforGeeks*. Retrieved February 14, 2021 from https://www.geeksforgeeks.org/huffman-decoding/

[23] 2017. Burrows - Wheeler Data Transform Algorithm. *GeeksforGeeks*. Retrieved February 14, 2021 from https://www.geeksforgeeks.org/burrows-wheeler-data-transform-algorithm/

[24] 2017. Inverting the Burrows - Wheeler Transform. *GeeksforGeeks*. Retrieved February 14, 2021 from https://www.geeksforgeeks.org/inverting-burrows-wheeler-transform/

[25] 2020. Seam Carving Algorithm: A Seemingly impossible Way to Resize an Image. *Analytics Vidhya*. Retrieved February 14, 2021 from https://www.analyticsvidhya.com/blog/2020/09/seam-carving-algorithm-a-seemingly-impossible-way-to-resize-an-image/

[26] Seam Carving. Retrieved February 14, 2021 from http://cs.brown.edu/courses/cs129/results/proj3/taox/

[27] Tech-Algorithm.com ~ Nearest Neighbor Image Scaling. Retrieved February 14, 2021 from https://tech-algorithm.com/articles/nearest-neighbor-image-scaling/

[28] What is bilinear interpolation? *The DO Loop*. Retrieved February 14, 2021 from https://blogs.sas.com/content/iml/2020/05/18/what-is-bilinear-interpolation.html

[29] Bilinear Interpolation - an overview | ScienceDirect Topics. Retrieved February 14, 2021 from https://www.sciencedirect.com/topics/engineering/bilinear-interpolation

[30] Discrete Cosine Transformations. Retrieved February 14, 2021 from https://datagenetics.com/blog/november32012/index.html

[31] Fractal Image Compression. Retrieved February 14, 2021 from https://users.cs.northwestern.edu/~agupta/_projects/image_processing/web/FractalImageCompression/

[32] AN INTRODUCTION TO FRACTAL IMAGE COMPRESSION. 20.

[33] Huffman Coding Algorithm. Retrieved February 14, 2021 from https://www.programiz.com/dsa/huffman-coding

[34] manassra/LZ77-Compressor. *GitHub*. Retrieved February 14, 2021 from https://github.com/manassra/LZ77-Compressor

[35] Lossless Data Compression: LZ77. Retrieved February 14, 2021 from https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossless/lz77/example.htm

[36] Data Compression - LZ-78. Retrieved February 14, 2021 from http://www.stringology.org/DataCompression/lz78/index_en.html

[37] Lempel-Ziv Compression Techniques. 9.

[38] Sadat97/Lz78. *GitHub*. Retrieved February 14, 2021 from https://github.com/Sadat97/Lz78

[39] COS 226 Burrows-Wheeler Data Compression Algorithm. Retrieved February 14, 2021 from https://www.cs.princeton.edu/courses/archive/spr10/cos226/assignments/burrows.html

[40] chaitan94/knit. *GitHub*. Retrieved February 14, 2021 from https://github.com/chaitan94/knit