Transilvania University of Brasov
- University of Mathematics and Computer Science -

Image Compression Using Quadtree Decomposition

Dobre Cosmin
Lopataru Mihnea

May 2022

Contents

1 Abstract

Memory has always been a discussing topic in the field of computer science. Ever since the first computer was invented, we constantly improved its capacity in order to be able to store more and more data. As a result, we went from a couple of bytes of storage to terabytes or even petabytes.

Of course, when memory modules got larger, so did the size of the data we wanted to store on them and this phenomenon can be very easily observed in photographs. As our demand for greater detail and higher quality images increased, so did their memory sizes. Take for example a **FHD (1920x1080)** picture, which only takes about **400KB**. Now compare it with a 4k or even 8k picture which can be a few good MB. This seems like a trivial problem, since our computers can store thousands of gigabytes of information, but when you think that the average person doesn't have a single photograph in their computer or phone, but rather thousands, or for more passionate people, hundreds of thousands, we can see how a terabyte of memory doesn't seem so much now.

This is where **image compression algorithms** come into play. With their help, the image size can be reduced, with a minimal impact on its aspect. Of course, as compression rates get higher, the degradation of the image becomes noticeable. **Compression** is done for storage purposes, as most devices right now can't fully render a large scale image, so in order to save space and optimize the size occupied by the photograph, techniques are used to remove details that the user wouldn't be able to notice anyway.

In this project, we will present what **quadtrees** are, why are they used for such applications and how we implemented them. We will also have a section in which we will talk about how the **compression process** was optimised, as the algorithm can be very costly from a time and memory point of view, so special programming techniques were used to speed up compression.

2 App Description

When making an application, it is very important to create a user-friendly and easy to use interface. Since all interactions are done trough the **GUI**, it has to be modern looking, with inviting and calming colors that gives the user a good experience. Using dark colors or colors that do not harmonise each other, could strain the user's eyes, creating discomfort and thus, leading to not wanting to use our app.

2.1 General Presentation

The app features a fully working GUI developed using windows forms, from which the user can choose different actions using the menu provided by the app.

Menu options:

- **Load image** → used to load an image (formats accepted: **.jpg, .jpeg and .bmp**) from the user's computer to the app
- **Compress image** → used to start the compression process
- **Save image** → used to save the compressed image to the user's computer (in **.jpg** format)

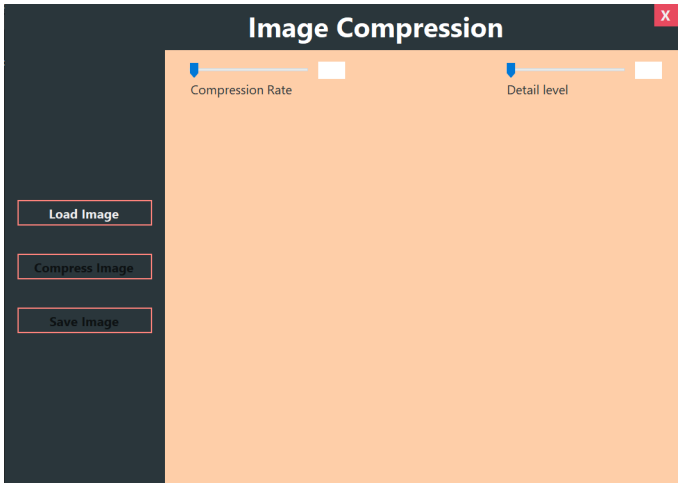


Figure 1

Each of the buttons described above activate based on the current state of the app, in order to prevent action conflicts and to compensate for possible user errors, ensuring the correct running of the application.

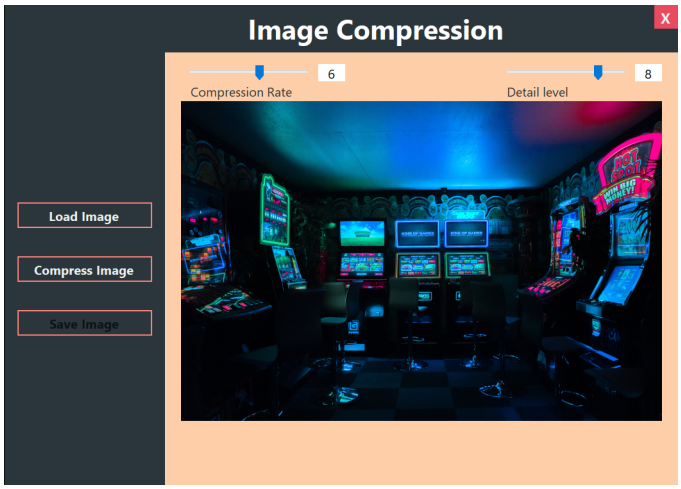
The user is also met with two sliders from which he can customise the compression factor based on his needs:

- **Compression rate** → dictates how much the image size is reduced (higher value ⇒ compression becomes visible)
- **Detail level** → dictates how much detail is kept in the compressed image (higher value ⇒ more detail is kept from the image)

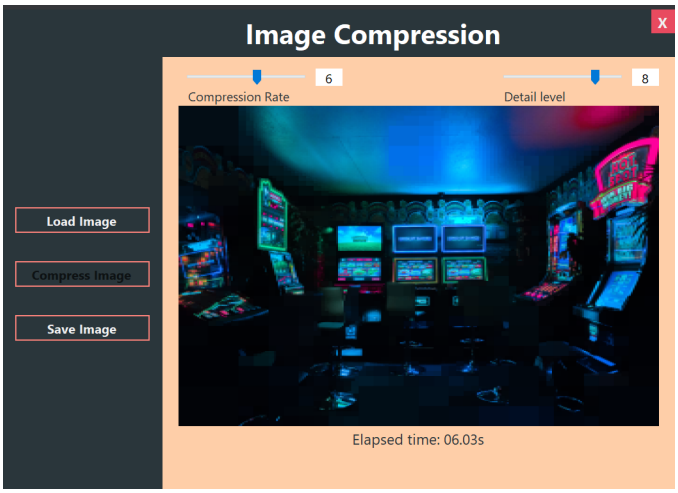
Figure 2

Once the user has loaded an image, it is displayed in the empty area seen in **Figure 1**. Once the user chooses to start the compression process, a loading bar will appear indicating the progress of the algorithm; once it is done the compressed image will be displayed in place of the original one, alongside a text which indicates how much it took to compress the image. This information is useful for seeing how image size affects the running times of the algorithm.

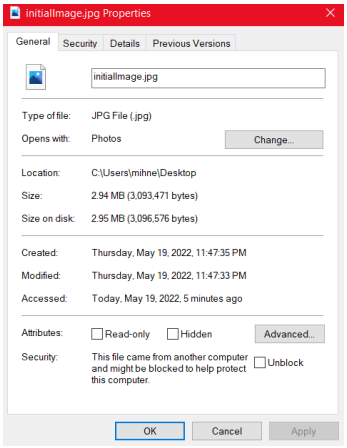
Photo by Carl Raw on Unsplash



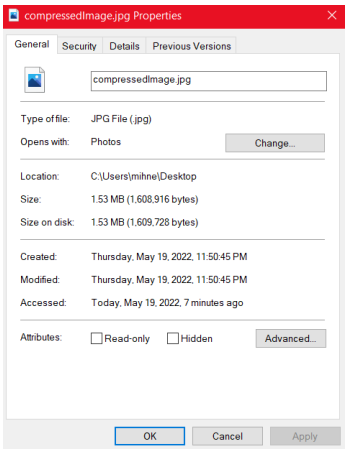
Initial image



Compressed image



Original size



Compressed size

If we closely look at the second photo, we can start seeing "pixel chunks" meaning our compression has worked and certain areas of the image have been optimised by removing unnecessary detail. As a result, the compressed image will have a smaller memory footprint than the original one.

At first it may seem like a small improvement, we reduced $\approx$ **1.5MB**, but remember that 1.5MB saved on thousands of pictures can add up to a significant amount. Of course, if we want, we can increase the compression rate and decrease the detail level to reduce the used memory even more. We chose this values because the impact on the image and quality was minimal.

2.2 Technology

Usually, an app like this would be developed using programming languages that have a fast compiling time. Since we used windows forms, our application was programmed in **C#**, using object oriented programming. Due to the fact that this programming language is harder to compile than say, Python or C, we had to come up with new methods to speed up the compression process. This techniques will be discussed in the future chapters.

The framework on which our application was built is **.NET framework**, more precisely, the better and improved version of it, **.NET Core**. The version chosen for the framework was **.NET Core 6**, as it has new and improved features which speed up development. It also features a series of optimisations that make the app run faster and smoother.

One other important feature that made us choose **.NET Core 6** is the fact that it is cross-platform. This means that our app can run on different machines, no matter the system architecture or operating system, making our application more accessible and user friendly.

2.3 Algorithm

Our compression algorithm employs the use of quadtrees to achieve its goal. More precisely, it does a quadtree decomposition on the image, in order the find the areas of the picture that can be filled with a single color. By doing so, large areas of a photograph that contain different shades of a color, can be replaced with an average one, reducing the information needed to store the image while also having a minimal impact on the image aspect.

After the algorithm finished its run, we will be able to see chunks of pixels appear in the image, giving it a more pixelated look. This means that the run was successful and the compression worked. Also, based on the level of detail present in the image, we will see that certain areas that require more attention to detail have smaller chunks. This is done, as we have mentioned previously, to help maintain as much of the initial aspect as possible.

Photo by Joel Filipe on Unsplash

Figure 1: **Original image**

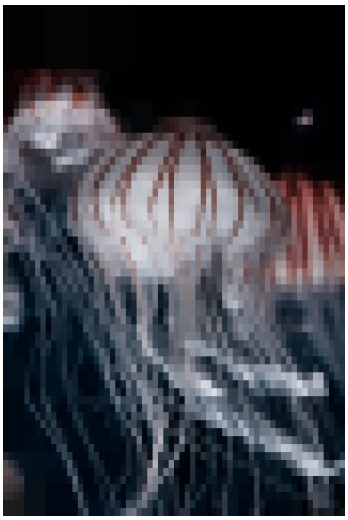


Figure 2: **Compressed image**

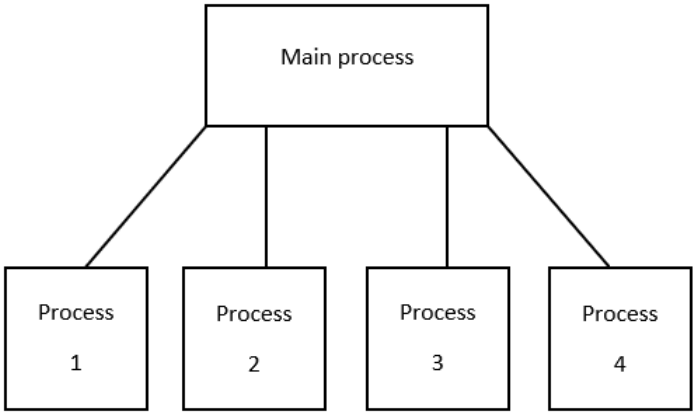
As we stated previously, we can clearly see that for the objects in the background, less detail is needed, so the pixel chunks are larger, whereas the jellyfish in the foreground is much more detailed and we need to be careful when splitting it, in order to preserve the original aspect. As a result, our algorithm divides it in smaller chunks in order to keep an acceptable level of detail.

2.4 Optimizations

Once our application was completed, everything was working very well and smoothly. However, once we started compressing bigger images (say 4k or even 8k resolution), our application was starting to show its weakness and that was time. For FHD pictures, we were getting roughly 3-4 seconds of wait time, however, for higher resolutions (depending on the level of detail present in the image) we would have to wait up to a minute to get a result.

This may not seem as a long waiting period, but if a user wants to use our application to compress hundreds of images, this quickly becomes a very serious problem. This is why, after completing the app, our focus shifted into bringing the compression time down as much as possible. This was done by applying a programming technique called multi-threading. In C# (and not only), multi-threading is a method used to achieve multitasking, meaning to solve different tasks simultaneously, decreasing run times.

To put it into practice, we used multiple execution threads, splitting the image compression process into four threads, each corresponding to one children of a node in the quadtree (as its name suggest, a node has 4 children resulting in 4 execution threads). This helped our application a lot, bringing down execution time from minutes (for high resolution images) to just a couple of seconds, lower resolution images getting into the milliseconds territory.



Dividing the main process into different processes that can be executed simultaneously

3 Quadtrees

In this part of our documentation, we will take a look at what quadtrees are and why we used them in the making of our application. We will also explain a specific type of quadtree, more precisely, region quadtree. We will not talk about other types, such as point-range quadtree or edge quadtree, as they were not relevant for our app.

3.1 What are quadtrees?

As Anthony D'Angelor presents it [2], "the quadtree is a hierarchical spatial data structure". More precisely, a quadtree is a complete tree data structure, having the property that each internal node has exactly four children. The reason they are categorized as a spatial data structure is because of their use to represent 2-dimensional space, performing different type of operations on it.

One of the most common application of quadtrees is splitting a given space into **four equally sized zones**, each node from the tree hierarchy holding the specific information of that zone. This can be done recursively, until a desired result is achieved or we have reached a maximum depth, if one exists.

Thanks to their properties and ability to represent a given space, quadtrees have cemented their role in the field of image processing, being a very reliable and useful tool whenever we need to get information or perform different operations on a photograph. The algorithm we used to compress our image is also known as quadtree decomposition, and in the following section we will see what it is based on and how it works.

3.2 What is quadtree decomposition?

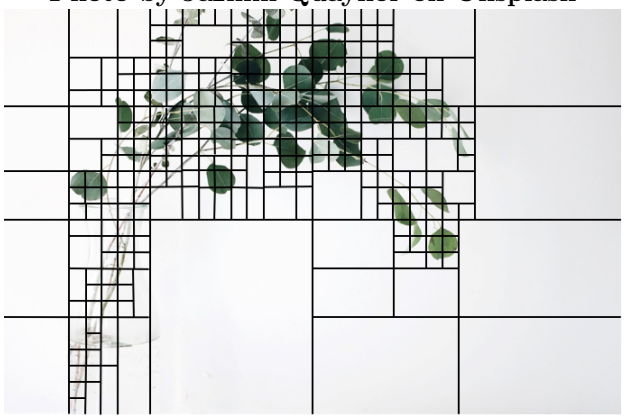
Quadtree decomposition is an image analysing technique used in image processing to obtain certain information about a photograph. This is done by subdividing a given image into smaller block, as presented in [9], "more homogeneous than the image itself".

Steps:

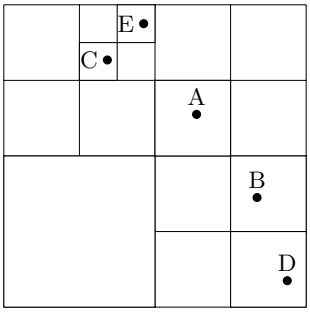
1. Analyse given image
2. If necessary divide the current square (at start the image) into 4 equal squares
3. Analyze the resulting squares
4. If we can't obtain the necessary information we check to see if we can continue splitting and
    - repeat from step 2
    - stop if we can't divide any further

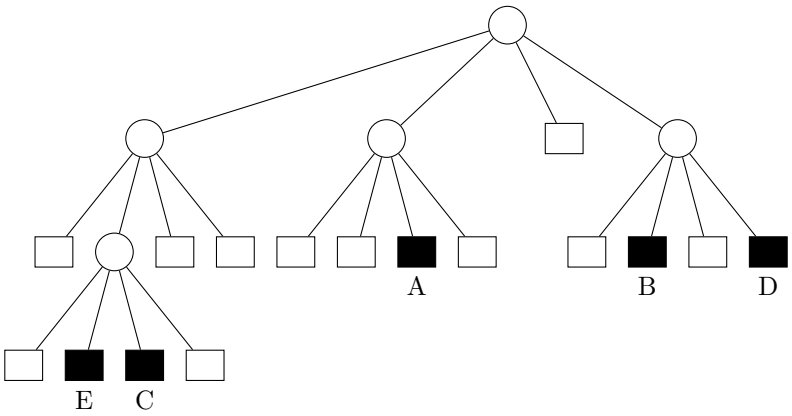
This process will leave us with a tree in which the further we go into, the more information we can find about the image. Of course, there are usually limits based on how deep a node can be, in order to avoid infinite cycles, but, the bigger the tree, the more detailed the data about the image is. We can see in the illustration below, that as we reach parts with greater details, the more divided that area is.

Photo by Jazmin Quaynor on Unsplash



When the algorithm has finished its run, the information that we will be interested in, will be stored in the tree's leaves, as it is illustrated below:





We can deduce from this information that the compression algorithm works by building a quadtree, after which it rebuilds the image based on the colors and information stored in the leaves.

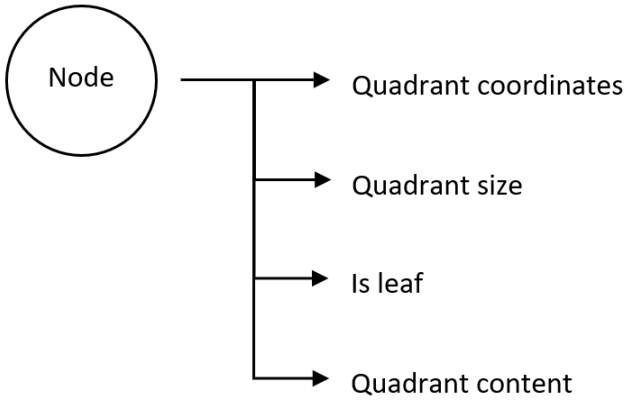
3.3 Region quadtrees

Region quadtrees are a very useful representation of a partition of space, in our case a **2D space**, being able to split it into four equal quadrants, subquadrants and so on until a stopping criterion is met. They are used to achieve the quadtree decomposition phenomenon.

We have stated in the beginning of this chapter that the quadtree is a complete tree data structure. **Region quadtrees** deviate from this definition, as a certain node can either have exactly four children or be a leaf, but tree completion is not guaranteed in this scenario.

Another difference between region quadtrees and normal quadtrees is the information a node is holding. In the case of region quadtrees, a node will usually contain various fields such as:

- information about the quadrant position
- information about a quadrant
- a field indicating if the node is or is not a leaf
- various spatial information and data about the quadrant it is referring to



We can observe that region quadtrees are based on the normal quadtrees, with slight modification in order to be better fitted for space divisions. This makes them a very powerful tool for image decomposition, which is exactly what we needed for our application. In the following chapter, we will dive a bit more into detail about the contents of our quadtree and what each node represented for our final image.

4 Image compression

In the previous chapters we gave a brief explanation about what quadtrees are and how we can use them. Now we will take a closer look at our implementation and the changes we brought to the classic algorithm.

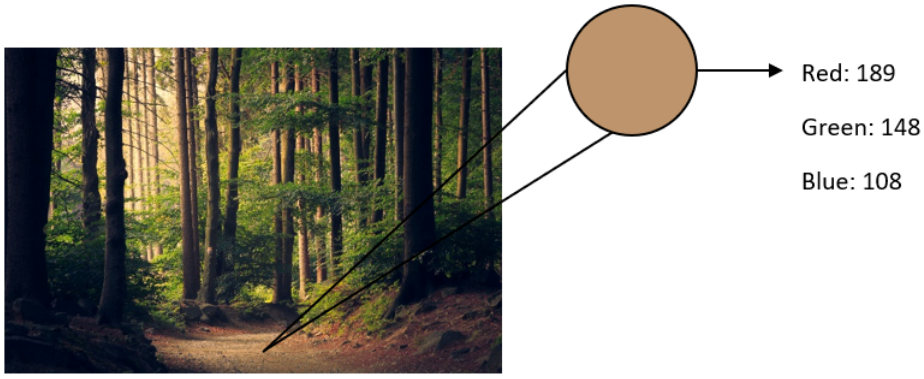
On a quick glance, **the idea of compressing an image** may seem pretty easy, just reduce memory by removing some details from it. The complicated part is how we define **detail**, the parameters which tell us how much we can degrade a certain part of the image. Another challenge is that for a computer, an image doesn't mean anything, but it is rather seen as a bunch of numbers which convey information about a certain area, so an important part of a compression algorithm is understanding **how an image is seen by a computer** and what each value present in it means, in order to be able to control it.

4.1 How is an image represented?

When we talk about computers, they only work with numbers, for them, images are just some values without any meaning, but rather, it is the programmer's role to assign a meaning to each number in order to obtain a certain result. In computer science, we can represent a picture in many ways, but the most common way and the one we used in our application is by referring to an image as **a bunch of coded color information**. More precisely, we see an image as a grid in which each cell (pixel) contains values representing a color.

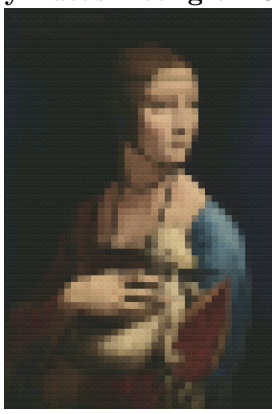
Colors are represented using **three channels**, also know as **RGB**, standing for red (channel), green (channel), blue (channel), each having a value between **0** and **255**, indicating that **channel's intensity**. Why this interval you may ask? Well it is because each color requires **one BYTE**, or **8 bits**, to be represented, resulting the range specified previously (0 to $2^8 - 1$).

Photo by Lukasz Szmigiel on Unsplash



We can see from the above illustration that an image is nothing more than **a matrix of pixels**, or to be more specific, a matrix of bytes, in which the color of each pixel present in the photograph is encoded in such a way so that the computer can work with it.

Photo by Lucas Hoang on Unsplash



Pixelated image that clearly shows each cell containing color information

Normally, to create a data structure from scratch that can do such things would be difficult, thankfully there already exists one and it is called **bitmap**. As it is put in [5], "a bitmap is one of many types of file formats for images stored in a computerized form"; in other words, **a bitmap** is a type of memory organization, used to store pixel information in a format that can be understood by the computer.

In **C#** we have a special bitmap class, implemented in the **System.Drawing library**, that has a variety of built-in functions that makes it easier for the programmer to manipulate and modify the pixels from an image.

```
OpenFileDialog fileDialog = new OpenFileDialog();
fileDialog.Title = "Select Image";
fileDialog.Filter = "Image File (*.jpg; *.jpeg; *.bmp; *.gif;) |*.jpg;
    *.jpeg; *.bmp; *.gif;";

if (fileDialog.ShowDialog() == DialogResult.OK)
{
    Bitmap image = new Bitmap(fileDialog.FileName);
    PictureDisplay.Image = image;
}
```

The above code was taken directly from our application and shows how we transfer the image loaded in our GUI (inside a picture box named PictureDisplay) and transforms it into a bitmap. A useful feature of the Bitmap class from C#, is it can work with **multiple file types**, as seen in the example above,

taking away from the programmer the struggle of handling different image extensions.

The Bitmap class that we used to handle images also comes with very useful functions that allows us to find the color of a pixel at a known location and also change it. Since the compression algorithm needs to be able to access a pixel and also change its attributes,these built-in functions make our lives easier.

Functions used:

- **GetPixel(int x, int y)** - returns the color of the pixel at the (x, y) coordinates
- **SetPixel(int x, int y, Color color)** - sets the pixel's color at the (x, y) coordinates with the color given as parameter

However, we were not able to take full advantage of this functions and this is because both **GetPixel()** and **SetPixel()** are very computational heavy, meaning they need a lot of time to execute. This didn't present an issue when we tested small images (720p or less), but as soon as we tried higher quality photographs, we ran into two major issues:

1. **execution times** were very big (minutes for an image)
2. **memory usage** was increasing while the application was running causing it to run out of available space

To solve this issue, we implemented our **own class to handle images**, being based on the Bitmap class from C#. Our version features optimizations to accessing and modifying pixels, drastically reducing execution time and memory needs.

The reason behind why **GetPixel()** and **SetPixel()** are not very fast, is that whenever they're called, they need to lock the bitmap into memory in order to be able to change it. Now imagine we need to call them thousands of times, constantly locking and unlocking a bitmap, eats up a lot of time. In our class, by creating a separate byte array, different from the bitmap. We are allocating from beginning the necessary space needed for our information and we can access it in **O(1) time**, which is a drastic improvement from the previous variant.

```
private void LoadPixelData(Bitmap image){
var rectangle = new Rectangle(0, 0, image.Width, image.Height);
var bitmapData = image.LockBits(rectangle, ImageLockMode.ReadOnly,
    image.PixelFormat);
IntPtr linePointer = bitmapData.Scan0;
var bytes = Math.Abs(bitmapData.Stride) * image.Height;
RGBdata = new byte[bytes];
System.Runtime.InteropServices.Marshal.Copy(linePointer, RGBdata, 0,
    bytes);
image.UnlockBits(bitmapData);
}
```

The above function was used to load all the information contained in the Bitmap in our new array. We used a **byte array** because, as we stated previously, the **RGB channels** are represented using 8 bits, so there is no need to use larger data types. In the end, we will get an array in which each cell represents a color channel value. Normally, this array could be divided into chunks of 3 elements (red value, green value, blue value), but because C# pixels are designed with 4 data channel, **ARGB (Alpha RGB)**, our so called "chunks" will be bigger than we initially thought. This isn't an issue, but rather something we need to be aware of, as it will influence the formula for the new **GetPixel** and **SetPixel** functions.

```
public Color GetPixel(int x, int y)
{
    var red = RGBdata[(x * 4) + (y * 4 * Width) + 2];
    var green = RGBdata[(x * 4) + (y * 4 * Width) + 1];
    var blue = RGBdata[(x * 4) + (y * 4 * Width) + 0];

    return Color.FromArgb(red, green, blue);
}
```

```
public void SetPixel(int x, int y, Color color)
{
    RGBdata[(x * 4) + (y * 4 * Width) + 2] = color.R;
    RGBdata[(x * 4) + (y * 4 * Width) + 1] = color.G;
    RGBdata[(x * 4) + (y * 4 * Width) + 0] = color.B;
}
```

As we can see, to access a pixel that would have been at the (x, y) coordinates in the bitmap, we need to find a formula that works for a linear space, which is:

(x * 4) + (y * 4 * ImageWidth) + ColorChannelOffset

We can now see why knowing the information "chunks" have 4 elements comes in handy, because if we would have multiplied by 3 instead of 4, we would have gotten an incorrect result, causing a faulty run of the application.

4.2 Quadtree image decomposition

We presented in previous chapter, the general idea behind the **quadtree decomposition** process and how we can apply it on images. This is exactly how we implemented it in our application, meaning that we divide each section of the image into four equal quadrants until we don't need to extract any more detail from the current quadrant. This is done in order to preserve only the necessary information from the photograph.

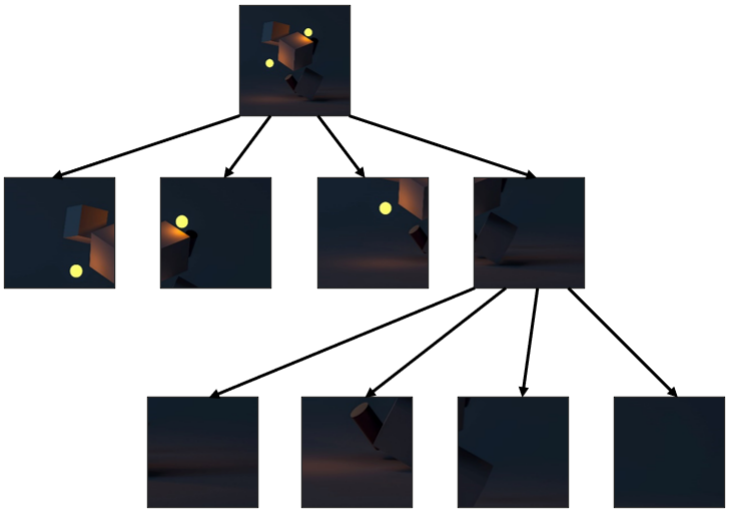
This process is the one that actually builds our tree, each node containing the following information:

- the coordinates of the quadrant
- if it is leaf or not
- the average color from the quadrant
- the quadrant's detail error
- the node's depth

At the end, we will have a quadtree in which each leaf contains the quadrants that can be optimised and the color with which we can fill the respective area. As a result, the deeper we go int our tree, the smaller the quadrants will be and the level of detail greater.

In the following figure, we have a graphical illustration of our algorithm, after which we will present it step by step. An observation that results from the bellow scheme is that the root of our tree will always contain the image itself.

Photo by Sebastian Svenson on Unsplash



...

Algorithm steps:

1. Initialize the root as being the original photo
2. Analyze the level of detail contained in the current node (in the beginning the root)
   - if the node has enough detail, we stop dividing
   - if the node does not have enough detail, we divide the section indicated by the node into 4 more subsections
3. Repeat this process until we can't divide anymore, or the maximum depth allowed for a node has been reached

```
while (queue.Count > 0)
{
    if (queue.TryDequeue(out var node))
    {
        //Find average color and level of detail error

        //Calculate total pixel count from the current zone

        //Calculate the color intensity for each channel (R, G, B)

        if (node.NodeError > detailTreshold && node.NodeDepth < maxDepth)
        {
            NotifyNewStep(ImageCompressionSteps.SplitNode);

            node.SplitNode();

            node.GetChildren().ToList().ForEach(queue.Enqueue);
        }
    }
}
```

In the following chapters we will take a look at how we find the **detail error** from a certain zone and how we know if we need to keep splitting the node or not. We will also present what the **"detail threshold"** and **"detail level"** roles are and the values they should have for the algorithm to provide a good solution.

4.3 Detail threshold

In the previous chapters, we have talked about detail and detail error a considerable amount of times. This is because, it is a vital component of our algorithm, without which it can't function properly, or stop for that matter.

The role of this **threshold**, is to know when we have extracted enough information from a certain zone. Of course, we will not get the full details out of it, how much being dictated by the "error" value . In other words, this component indicates how much we can "destroy" the image, or remove detail out of it, thus reducing memory size, any quadrant which isn't detailed enough being divided after.

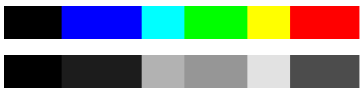
Usually, this value is fixed by the programmer, after many tests being relatively easy to find an interval in which the optimal results are achieved. However, this removes the freedom from the user to choose how much detail he wants to keep / remove. As a result, we decided to let the user choose this threshold.

After a value has been chosen, every time we start analyzing a quadrant, we calculate that specific zone's "**detail error**" and compare it to our threshold. If the quadrant's value is greater that the maximum error accepted, we divide it further. The level of detail from a certain zone is calculated using the bellow formula:

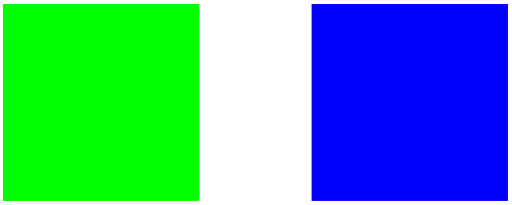
$$\textbf{error} = 0.2989 * R + 0.5870 * G + 0.1140 * B$$

This formula indicates how the average human eye perceives brightness, also called **color luminescence**. This percentages come from the way the eye sees color. Because we do not perceive colors in a linear way, a "greener" object will appear brighter to us than say, a blue one. "The role of luminance is to show this difference" [3].

A question that may appear is, why do we need brightness to find detail? Well, when we want to find the level of detail from a certain quadrant, we first convert the image contained in the respective zone to gray-scale. After that, we try to find the **average** gray color of that particular zone. This is done by taking the **weighted average** of each color, multiply it by a constant, and summing it all up, exactly what the above formula describes. What we will get after putting the values trough the equation mentioned above, is a shade of gray. If we tilt towards black or a darker shade (low brightness), it means there is still detail that can be extracted from the respective zone. If we get a lighter shade (high brightness), this means there isn't much detail left to extract from the image, and we can stop (we can see from the image bellow, that green represents a lighter shade of gray compared to blue or red).



We can see from the above image, that the intensity of the color green, becomes even more noticeable when it is mixed with other colors, as in our case, red, resulting yellow (yellow → R:255, G:255, B:0).



As we can observe, even though both colors have the same intensity (Green square(0, 255, 0) and Blue square(0, 0, 255)), the green square appears to the human eye as being brighter than the blue one. Bellow we have the implementation of the elements we talked about.

```
private static readonly double RedPercentage = 0.2989;
private static readonly double GreenPercentage = 0.5870;
private static readonly double BluePercentage = 0.1140;

var greenFrequency = FindColorFrequencies(results.Item2, Colors.Green);
node.NodeError = GreenPercentage * FindWeightedAverage(greenFrequency,
    total);

if (node.NodeError <= detailTreshold)
{
    var redFrequency = FindColorFrequencies(results.Item2, Colors.Red);
    node.NodeError += RedPercentage * FindWeightedAverage(redFrequency,
        total);
}

if (node.NodeError <= detailTreshold)
{
    var blueFrequency = FindColorFrequencies(results.Item2, Colors.Blue);
    node.NodeError += BluePercentage *
        FindWeightedAverage(blueFrequency, total);
}
```

A very important thing that we need to remember when dealing with this types of operations, is the term **"weighted average"**. This method takes into consideration the aspects mentioned previously (luminosity). In order to calculate such average, we will need to use a special data structure, called **histogram**.

In image processing, a histogram is a graph, which indicates the intensity of each pixel present in a certain image. Usually histograms are used for gray-scale images, taking the shape of an one dimensional array. However, for our use, we had to implement a 3D data structure, which has an axis for each color (Red, Green, Blue), as presented in [6]. To create a histogram, the method is quite simple. All we need to do is iterate through all the pixels in the image and using a matrix structure, count the pixel color combination we encounter. In other words, a histogram is, to a certain extent, a frequency array for the colors

present in the image.

When we implemented our histogram, we decided to use a **dictionary** (or an unordered map as it is in C++), in order to be able to access a certain color, as well as checking its existence, in O(1) time, making our algorithm faster.

```
Dictionary<Color, int> histogram = new Dictionary<Color, int>();

for (int i = startCorner.X; i < stopCorner.X; i++)
{
    for (int j = startCorner.Y; j < stopCorner.Y; j++)
    {
        Color color = image.GetPixel(i, j);

        if (histogram.ContainsKey(color))
        {
            histogram[color] = histogram[color] + 1;
        }
        else
        {
            histogram.Add(color, 1);
        }
    }
}
```

Once the histogram is created, we can now start to calculate the frequency of the **three main color channels (Red, Green, Blue)**, get their weighted average and then finally use the formula mentioned in the beginning and get our wanted error.

In the end, we can see that the process of finding this **"detail error"**, is very complex and requires a lot of time. However, there is no method to skip this part, as it is a vital component of the algorithm and must be calculated. There are ways, however, in which we can speed up the process, one of them being shown in the following code snippet:

```
node.NodeError = GreenPercentage * FindWeightedAverage(greenFrequency,
    total);
if (node.NodeError <= detailTreshold)
{
    var redFrequency = FindColorFrequencies(results.Item2, Colors.Red);
    node.NodeError += RedPercentage * FindWeightedAverage(redFrequency,
        total);
}
if (node.NodeError <= detailTreshold)
{
    var blueFrequency = FindColorFrequencies(results.Item2, Colors.Blue);
    node.NodeError += BluePercentage *
        FindWeightedAverage(blueFrequency, total);
}
```

The calculation of the **weighted average**, takes a lot of computational time, by calculating it only when necessary, we save the unnecessary calculations, thus speeding up the process. The logic behind this method is that we don't need to add all the values to find out if we need to divide the current zone or not, but rather keep adding them until we have passed our set threshold.

4.4 Detail level

When we talk about images, we constantly hear the word detail, how detailed an image is or the lack of detail in it. Well, in simple terms, **detail** represents how much information our eyes can perceive. Let's take for example a picture of a human face, as the one bellow.

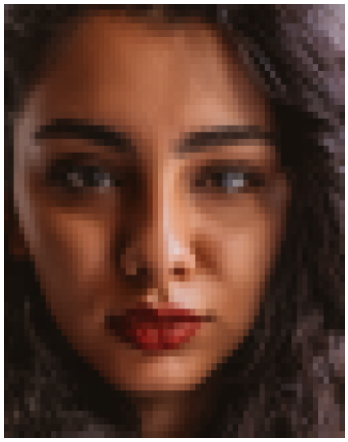
Photo by Omid Armin on Unsplash

If we look at the face, we would consider this picture as being very detailed, we can see the pores, skin texture, small hairs and the tiny imperfections of the face. Of course, in their absence, our brain can fill in the missing parts, but this process is not as satisfying as being able to see them, plus the missing parts are noticeable, aspects that are presented in [7].

Since we are dealing with image compression's, inevitably, some detail of the original image will be lost. Just as in the case of the default detail threshold value, usually this level is set by the programmer, in order to provide a certain result. Once again, we decided to give the user the freedom of choosing how detailed the image should be, the values ranging from 1 to 10, the higher the value the less detail will be removed from a certain area.

In code, this **detail level** translates to the **maximum depth of the quadtree**, forcing the division to stop if we were to go deeper than allowed. As we stated in the previous chapters, the deeper we go into a quadtree, the more

detail we find about a certain image. As a consequence, altering the depth of the quadtree will impact the quality of the final output.



Low detail (detail level = 6)

High detail (detail level = 8)

As we can see from the side by side comparison above, for the lower detail level, we start seeing pixel chunks and features like pores or fine eyebrow hairs are gone, whereas for the higher detail level, even tough some attributes are lost, we can still see the pores and fine hairs, skin texture being the one who is taking the main hit.

Detail level, even though is easier to implement and use than the detail threshold, it is just as important for our algorithm, its absence causing an infinite cycle of dividing and comparing. It enforces a limit which once exceeded, no matter the error value, the node will no longer be divided. It will also have a visible impact on the compressed image size.

```
if (node.NodeError > detailTreshold && node.NodeDepth < maxDepth)
{
    NotifyNewStep(ImageCompressionSteps.SplitNode);

    node.SplitNode();

    node.GetChildren().ToList().ForEach(queue.Enqueue);
}
```

As we can clearly see from the code sample above, the node will be split if the node's error is greater than our set threshold and if the node's level is lower than the max level permitted.

5 Optimizations

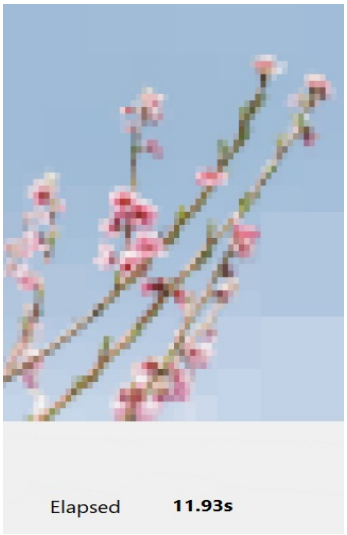
Optimizations are a vital part of any application, both **memory** and **time optimizations** being crucial for the proper working of the final product. However, in this day and age, memory is no longer that big of an issue, as our computers can handle billions of bytes of information. This is why, usually, we prefer using more memory, if that means faster execution times. In our case, we used **multi-threading** in order to speed up the running of our application.

5.1 Why was it necessary?

In the beginning of this documentation, we mentioned that one of the biggest issues we encountered while developing our app, was time. More precisely, the time the user had to wait in order to get the final result. In the chapters that followed, when we talked about how we implemented different parts of the algorithm, we mentioned various changes we made to the original code, optimising different operations.

However, the gains made from these optimisations were not big enough, as we only managed to shed a couple of milliseconds from the final time. While helpful, when the waiting time got into minutes, it was clear we had to come up with a new solution. That came in the form of execution threads, or threads for short.

The gains we had in terms of execution time were considerable this time. We managed to shrink the compression times from 1-2 minutes to roughly 10 seconds, which is a huge improvement compared to the non-thread implementation. Now, our application, not only can compress images, but can do it in a very short time, now being practical to use for a large number of images.



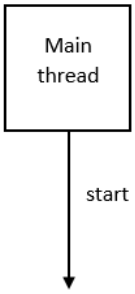
Before using threads

After using threads

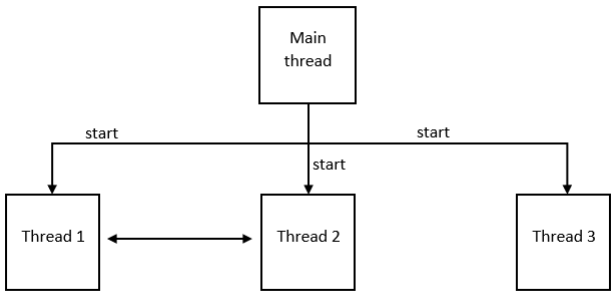
5.2 What are execution threads?

In the field of computer science, **a thread** is a "small set of instructions designed to be scheduled and executed by the CPU", quoted from [4]. This instructions are executed independently, without being affected by the parent process.

Today, most processors are **multi-threaded**, meaning they can work with multiple threads simultaneously. This is a very useful feature, as it enables us to multitask, completing multiple processes at the same time. Usually, the application has one execution thread, the main one, this means that whenever we run the program, a process can't start before the active one is completed. Usually, this is not a big issue, however, since processing a quadrant of an image needs a lot of time, waiting for a quadrant to be done processing to start the next one, can lead to massive waiting times. This is why being able to process multiple quadrants at once reduces waiting times drastically.



As we can see from the above illustration, in the case of a **single thread application**, a single thread is allocated and the principle mentioned above is applied.



In the event of a **multi-threaded application**, we can define new execution threads which take the load of the main one, this phenomenon is also know as **parallelism**. As a consequence, multiple processes can be executed at the same time, stopping a process's starvation.

In C#, in order to create a multi-threading application, we need to use the implemented class called **Thread**. This class encapsulates execution threads and offers for the programmer all the necessary functions to manage and handle them.

```
var quadrants = root.GetChildren();

ThreadStart first = delegate { new QuadTree(detailTreshold,
    maxDepth).SplitQuadrant(image, quadrants[0]); };
ThreadStart second = delegate { new QuadTree(detailTreshold,
    maxDepth).SplitQuadrant(image, quadrants[1]); };
ThreadStart third = delegate { new QuadTree(detailTreshold,
    maxDepth).SplitQuadrant(image, quadrants[2]); };
ThreadStart fourth = delegate { new QuadTree(detailTreshold,
    maxDepth).SplitQuadrant(image, quadrants[3]); };

Thread firstQuadrant = new Thread(first);
Thread secondQuadrant = new Thread(second);
Thread thirdQuadrant = new Thread(third);
Thread fourthQuadrant = new Thread(fourth);
```

When declaring a thread, we need to specify the function (process) which that thread will execute. This function needs to be either a static one, or a delegate (the C# equivalent of a pointer to a function).

A last aspect which we need to take into consideration when dealing with threads, is that one thread may finish its execution faster than the others. This doesn't seem like a big issue until we realise that the main thread may finish it's execution before a thread that is processing the image is done working. This will result in unpredictable and incorrect results. To solve this issue, we need to use the Thread member function **Join()**, which forces the threads to wait until all processes are completed.

```
firstQuadrant.Join();
secondQuadrant.Join();
thirdQuadrant.Join();
fourthQuadrant.Join();
```

Since modern-day processor all support multi-threading, it makes perfect sense why our application should take advantage of that. Not only do we speed up the compression process, but we also give the processor a bit of "breathing" room.

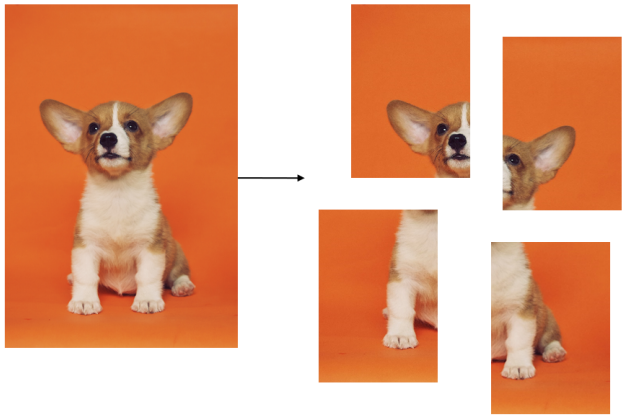
5.3 Using threads to speed up the compression process

We mentioned in the beginning of this chapter that, despite our improvements on the normal code, the app was not fast enough, thus demanding the use of threads. This was necessary because the time needed to analyze a quadrant, calculate its error and perform all the necessary operations, is very big.

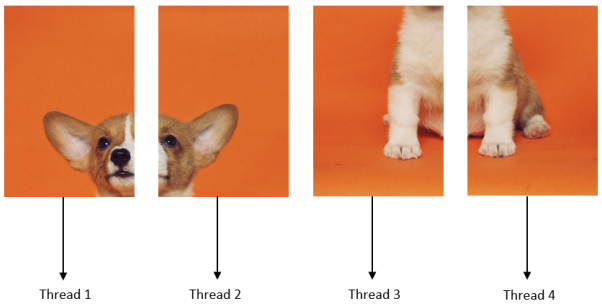
If we look at the code samples in the previous subsection, we can see that the approach we took, defines **4 executions threads**, one for each quadrant. More precisely, we know that the root will contain the image as a whole. This image will automatically be split into 4 quadrants, so for each quadrants resulting from this split, we assign a thread that will start dividing and compressing from that point onward.

For a better understanding, here is a visual representation on how the image is split and the tasks assigned to the threads.

Photo by Alvan Nee on Unsplash



Splitting the initial image into four equal quadrants



Assigning each quadrant a thread that will compress it

5.4 Time differences

We talked about the improvements that **multi-threading** brought to our application and how much we managed to **reduce compression times**. In the following examples, we will have a series of comparisons between the non-threaded application and the threaded one.

Test 1 (1920 x 1280)

Photo by Agê Barros on Unsplash

Without using threads

Using threads

Test 2 (2400 x 3000)

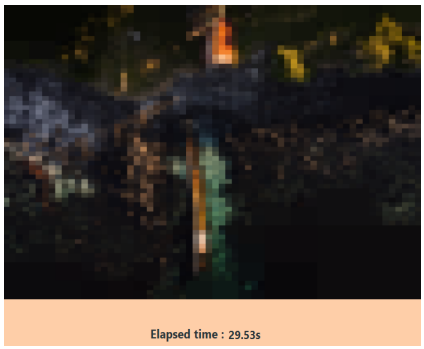
Photo by Tim Huyghe on Unsplash

Without using threads

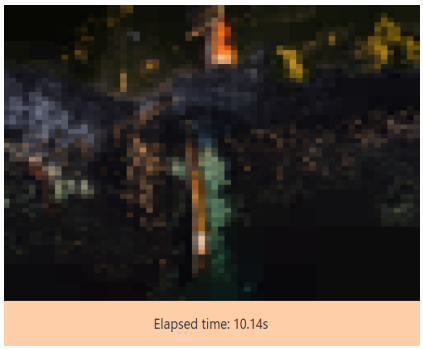
Using threads

Test 3 (8000 x 5000)

Photo by Nathan Jennings on Unsplash



Without using threads



Using threads

All the tests were done using the **same detail threshold** and **detail level**, on the same machinery. We can see that for low resolution images, the gain is substantial but not necessarily mind blowing. However, as our images get larger, we can see drastic changes (**up to 20 seconds faster**).

6 Conclusions

The purpose of this documentation, was to shine a light into what **image compression** consists of and the basic notions of **quadtrees** and **the algorithms** which uses them.

Troughout this paper we could see how useful this data structure can be, the variety of implementations and the power it holds, being able to represent 2D spaces and divide them. This is the reason why quadtrees cemented their role into image processing, especially compressions, when we need to modify areas of the image in order to reduce memory size.

Last but not least, we also covered, in a superficial manner, the basic notions of **thread programming** and **parallelism**. We could see how something like threads can make a massive difference in terms of execution times, especially since image processing is a very time consuming and CPU demanding task. By using **multi-threaded programming**, we allow the CPU to "breath" freely, while also reducing computational times drastically.

References

[1] Chet Chopra. *A Brief Intro to Multi-Threaded Programming*. Medium, 2019.

[2] Anthony D'Angelo. *A Brief Introduction to Quadtrees and Their Applications*. Canadian Conference on Computational Geometry, 2016.

[3] Wayne Fulton. *A little more detail about Histograms*. Scantips, 2010.

[4] Computer Hope. *Thread*. Computer Hope, 2019.

[5] R. Kayne. *What is a Bitmap Image?* EasyTechJunkie, 2022.

[6] R. Fisher, S. Perkins, A. Walker, E. Wolfart. *Intensity Histogram*. HIPR2, 2003.

[7] Josh Rose. *Understanding Detail in Photography*. Medium, 2017.

[8] Daniel Shiffman. *Images and Pixels*. Processing, 2022.

[9] Math Works. *Quadtree decomposition*. Math Works, 2022.

[10] Tanner York. *Quadtrees for Image Compression*. Medium, 2022.