OCR SUDOKU

The project report

Barbora Plašovská

Alexandre Bailly

Cheikh Christopher Enrique Tarabay

Elsa Keirouz

EPITA

December 2021

Table of content

1 Introduction

1.1 Our group

Our group is "**ab2e**" and it is composed of **Alexandre** Bailly, **Barbora** Plasovska, **Elsa** Keirouz and Cheikh Christopher **Enrique** Tarabay.

1.2 Progress

In order to be as efficient as possible, we divided our tasks :

- **Elsa** took care of solving the sudoku, creating the gui, and linking all the different parts together.
- **Barbora** was in charge of the neural network, so she mainly focused on character recognition and grid reconstruction.
- **Enrique** worked on loading the image, greyscale, binarization, rotation, contrast enhancement, and noise cancelling,. The algorithms were greatly improved from the first defence since now it is able to process images more accurately than before and is more efficient.
- **Alexandre** focused on image-splitting as well as grid detection.

As it can be seen in the table in the "Progress" section, every group member had an assistant if needed. However, it is important to note that we have all been helping each other, which is crucial for the success of our project.

1.3 Obstacles

As this project is rather challenging to execute, it was inevitable for us to run into a few problems while working on our respective parts.

Elsa: She had a bit of trouble figuring out how to work with pointers to a two-dimensional array. She also encountered a few issues while working with strings, notably while manipulating files, but she came up with solutions such as creating a new char array and adding all the elements into it. Finally, she found adding an argument to the main function a bit confusing at first, but after looking it up online and reflecting back on the second C practical, she figured it out. Obviously, all of these obstacles have been overcome and she now has a better understanding of all of these notions.

During the second wave of work, Elsa designed the graphical user interface, and worked on linking the different parts together. She had a bit of trouble figuring out the right format at first, but she ended up working with GtkFixed and GtkStack, and everything worked out.

In order to be able to use the solver to create the final solved grid, the pointer to the two-dimensional grid was changed into double pointers.

Barbora: When she started working on the neural network, she was very confused with all of it and how to actually code it mainly because we are coding in C. In S2, we already did a bit of AI in one of the practicals, where we used the biological algorithm to train our flappy birds to jump infinitely. Back then, she didn't really understand how it was supposed to work, but looking back at it now, it has helped her a lot with the implementation of the xor function for the neural network. One of the biggest obstacles in C was definitely the absence of classes. One of her first drafts of the neural network didn't have any structs, because she didn't know about them. The fun really began when she added structs and pointers. She struggled to understand how to work pointers with two-dimensional arrays, but in the end she did manage to make it work. Furthermore, when she started to train the neural network to recognize the digits, she struggled a lot since on her mac she cannot have the SDL library.

Enrique: He had trouble figuring out which libraries should be used for image processing. However, after working on our third practical, he realised that using the sdl library was the best solution. Then, it was a matter of deciding between sdl1.2 or sdl2. The main differences were sdl2's more advanced features, which were not necessary for image processing, and would need several modifications to be able to compile successfully, which is why he ended up using sdl1.2. After that, everything fell into place. For grayscale, it was pretty straightforward and all that was needed was a simple equation to compute using the RGB values. Turning the image to black and white was a bit more complicated since it needed a threshold which varies depending on the image rgb values. To make it work, he had to either find a good algorithm that works for most cases or work with trial and error. Since he was determined on applying a good algorithm for the binarization, a lot of research had to be done for how image processing is done. The two most reliable algorithms he found for this were binarization using mean filter and binarization using median filter. After testing both, the median filter worked best. It was then necessary to figure out how manual rotation was done. He was inspired by a lot of articles explaining how a rotation matrix works and applied it with the help of the built in math library.

Alexandre: One of the first obstacles he encountered was finding a concrete way to detect a sudoku grid. Online, he could only find commonly used algorithms, but he had trouble understanding them. To remedy this problem, he decided to try and implement another method based on the assumption that the sudoku grid would represent a significant part of the picture (at least 3/8). Instead of using a blob algorithm or a Hough transformation algorithm to find the grid, he decided to create a function which would be able to detect a big square (the sudoku grid), then save the coordinates of the 4 corners to use probabilities to find the numbers located inside it. He succeeded in doing so. However, the numbers that it gave out were not of the right size and had black outlines. Therefore, to fix this problem, he had to

create a function that would remove the black outline to only keep the number and then transform the number into a 28 by 28 pixel image.

As a group: We haven't encountered any issues of any form as a group. (Except for the occasional inevitable merging issues!) … We just had trouble figuring out what the makefile should look like but it turned out to be pretty easy to set-up once we understood it.

2 Presentation of the project

2.1 Technical aspects

2.1.1 Solver

Having already implemented a Sudoku solver in C#, **Elsa** was already familiar with backtracking algorithms. That was a huge asset in her favor, as she already had a clear idea of the different functions that she needed to create and how they were all tied together.

She started by creating a two-dimensional int array of size 9*9 called grid, and designating a pointer to its very first element.

Two-dimensional array and pointer.

```
int grid[9][9];
int* p = &grid[0][0];
```

She then created a load function which takes the path to the file containing the grid as a parameter, and fills up the two-dimensional array accordingly by parsing through it.

```
void load(char path[])
{
    char c;
    int x = 0;
    int y = 0;
    FILE* f = fopen(path, "r");
    while ((c = fgetc(f))!= EOF)
    {
        if ((c >= '1' && c <= '9' )|| c == '.')
        {
            if (c == '.')
                c = '0';
            *(p + y*9 + x) = c - '0';
            x+=1;
        }
        if (x == 9)
        {
            y += 1;
            x = 0;
        }
    }


    fclose(f);
}
```

Grid-loading function.

After that, she wrote 2 types of functions :

- *is_element_solved*: checks whether the line, column, or square, situated at the given coordinates (passed as a parameter) is already solved (every number between 1 and 9 inclusive appears exactly once). She also wrote a function to check if the whole sudoku is solved by going through all the coordinates and checking lines, columns and squares.
- *already_in_element*: checks whether a given number between 1 and 9 inclusive is already present in the given line, column, or square.

```
int is_solved()
{
    for (int i = 0; i < 9; i++)
    {
        if (!is_column_solved(i) || !is_line_solved(i) ||
        !is_square_solved(i / 3, i % 3))
            return 0;
    }
    return 1;
}
```

Function that checks if the sudoku is solved.

Next up is the solver, a recursive function _solve which calls all the previously implemented functions. It takes the coordinates (x,y) and does the following :

- If y>= 9 (outside of the grid), it returns 1, meaning that the grid is solved.
- Else, it sets the next coordinates using the function SetNextCoordinates and checks for the given x and y. If the value at the position (x,y) is different than 0, the function returns a recursive call for the next coordinates (x+1,y). If not, it checks which value between 1 and 9 fits and puts it at the position (x,y) then proceeds to solve for the remaining coordinates recursively according to that value: if it works out, then the function returns 1-indicating that the grid is solved- if not, then it sets the value at (x,y) back to zero and tries for the next digit.
- If none of the values work, the function returns 0, indicating that the sudoku wasn't solved.

Being a recursive function that needs an initial call, **Elsa** wrote a hat function called solver which takes no parameters, calls _solve with parameters (x,y) = (0,0) , and returns nothing.

All of these functions were implemented using pointers. In order to access grid[y][x], she used the formula *(p + y*9 +x).

2.1.2 Saving the grid

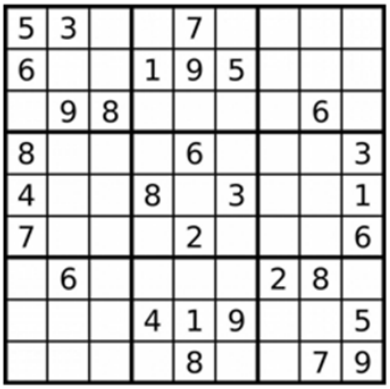
For saving the solved grid, Enrique and Alexandre made a function taking two 9 by 9 double pointers as a parameter which are used in the form of a matrix. The first matrix represents the old matrix and the second matrix represents the new matrix. For the sudoku grid, he emptied a sudoku grid using a picture editing software called paint.net and created 9 images representing the values 1 to 9 colored white and 9 images representing the values 1 to 9 colored green. It compares the old matrix and the new matrix in order to identify whether there was a change and assigns the color accordingly.

2.1.3 Image Processing

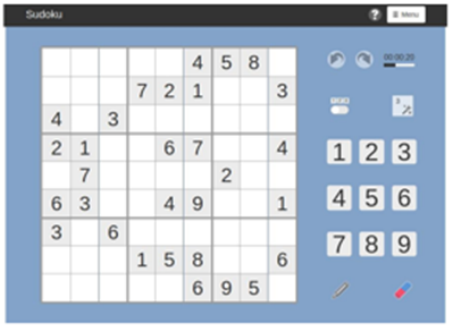
2.1.3.1 General Explanation

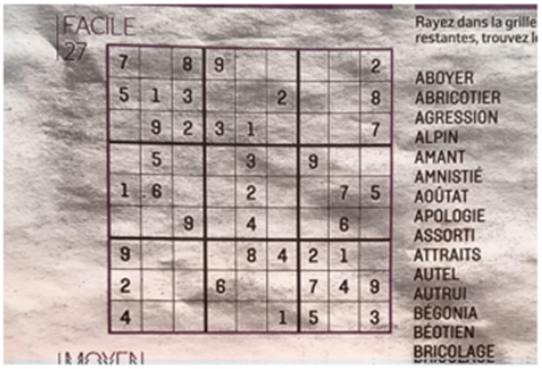
Enrique worked with the SDL library and its built-in functions in order to process the images. The trick was that the pixel information is retrieved in the form of a hexadecimal: 0x00000000. He needed to separate the values for red, green, blue, and alpha which are represented respectively by every pair of digits after the ‘x’. By looping through the image’s width and height, editing the pixels accordingly was a good starting point.

These are the 6 images we will use as example for the following parts:




MOYEN
28
BÉGONIA
BÉOTIEN
BRICOLA
BUNGAL
BUTANIE
CABANE
CAPTATI
CENTAIN
COSY


Sudoku
Menu

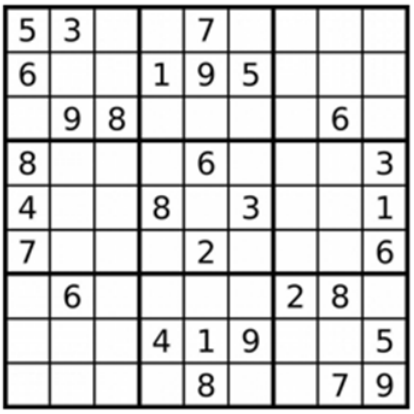

FACILE
27
Rayez dans la grille
restantes, trouvez l
ABOYER
ABRICOTIER
AGRESSION
ALPIN
AMANT
AMNISTIÉ
AOÛTAT
APOLOGIE
ASSORTI
ATTRAITS
AUTEL
AUTRUI
BÉGONIA
BÉOTIEN
BRICOLAGE

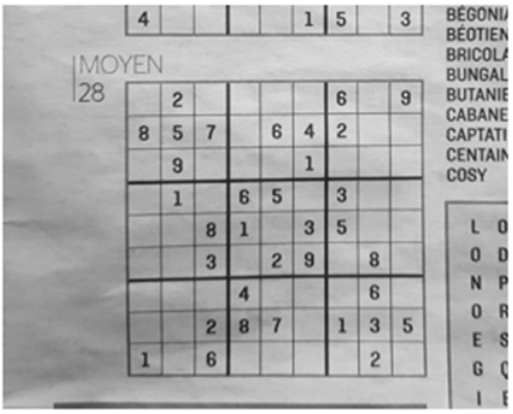
2.1.3.2 Greyscale

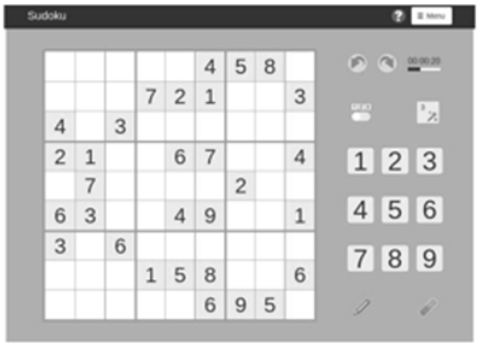
The challenge for greyscale was finding the best algorithm for processing the image rather than one which is good for looks. The three methods he looked into were the lightness method, average method, and luminosity method. The lightness method uses the most prominent and least prominent colors of the image by doing: (Most Prominent + Least Prominent) / 2. The average method adds the r,g,b values of each pixel and divides them by 3.
After testing both of these algorithms, none were good enough. The problem with the lightness method was that the contrast is usually reduced which makes the grid less visible. The average method almost does nothing to make the image more visible to be able to detect the grid so that was disregarded too. The only logical method left to use was luminosity.

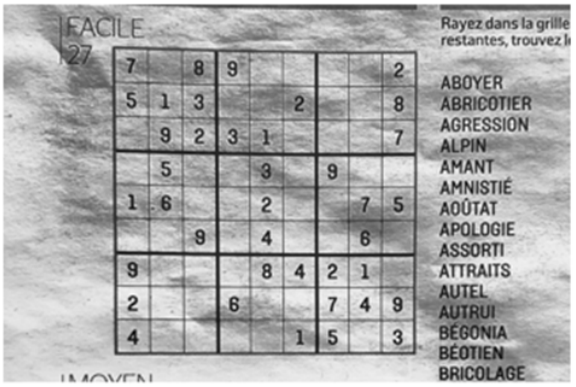
The equation he used was: 0.3 * r + 0.59 * g + 0.11 * b.

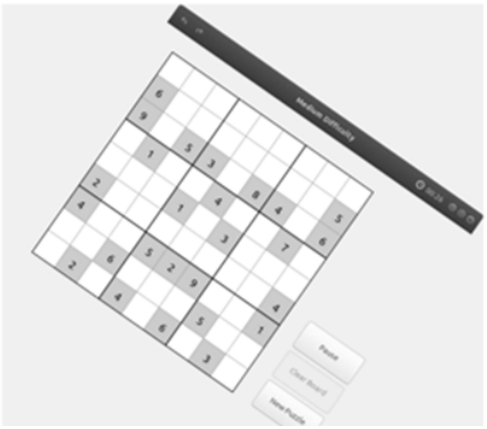
```
void ApplyGreyscale(SDL_Surface *surface)
{
        int width = surface->w;
        int height = surface->h;

        for (int x = 0; x < width; x++)
        {
                for (int y = 0; y < height; y++)
                {
                        Uint32 pixel = get_pixel(surface, x, y);
                        Uint8 r, g, b;
                        SDL_GetRGB(pixel, surface->format, &r, &g, &b);
                        Uint32 average = 0.3 * r + 0.59 * g + 0.11 * b;
                        r = g = b = average;
                        pixel = SDL_MapRGB(surface->format, r, g, b);
                        put_pixel(surface, x, y, pixel);
                }
        }
}
```




MOYEN
28
BÉGONIA
BÉOTIEN
BRICOLA
BUNGAL
BUTANIE
CABANE
CAPTATI
CENTAIN
COSY


Sudoku
Menu


FACILE
27
Rayez dans la grille
restantes, trouvez l
ABOYER
ABRICOTIER
AGRESSION
ALPIN
AMANT
AMNISTIÉ
AOÛTAT
APOLOGIE
ASSORTI
ATTRAITS
AUTEL
AUTRUI
BÉGONIA
BÉOTIEN
BRICOLAGE


Medium difficulty
Pause
Clear Board
New Puzzle

2.1.3.3 Contrast Enhancement

Before turning the image to black and white, he needed to check whether a contrast enhancement was needed. To do that, he created a function which calculates the variance of the whole image.

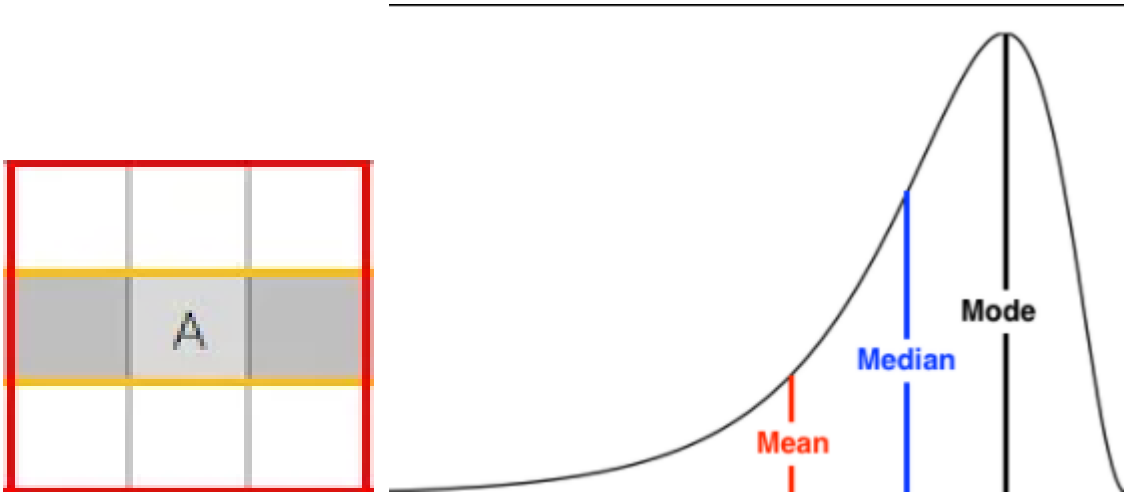
```
static inline
float ApplyEnhancement(SDL_Surface *surface) // returns variance
{
    int width = surface->w;
    int height = surface->h;
    int n = width * height;

    float sum = 0;
    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            Uint32 pixel = get_pixel(surface, x, y);
            Uint8 r, g, b;
            SDL_GetRGB(pixel, surface->format, &r, &g, &b);
            Uint32 average = (r + g + b) / 3;

            sum += average;
        }
    }

    float mean = sum / (width * height);

    float sqDiff = 0;
    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            Uint32 pixel = get_pixel(surface, x, y);
            Uint8 r, g, b;
            SDL_GetRGB(pixel, surface->format, &r, &g, &b);
            Uint32 average = (r + g + b) / 3;

            sqDiff += ((average - mean) * (average - mean));
        }
    }

    return sqDiff / n;
}
```

Now that he is able to determine whether the contrast of the image needs to be increased or not, he needs a function to enhance the contrast. The calculation for the contrast enhancement could be done either using global thresholding which calculates the threshold once for all pixels or using local thresholding which calculates the threshold of every pixel individually based on a filter size . After testing both, he determined that for this one enhancement using global thresholding would be enough since the binarization is going to use local thresholding. Hence the function applied for contrast enhancement was the following:

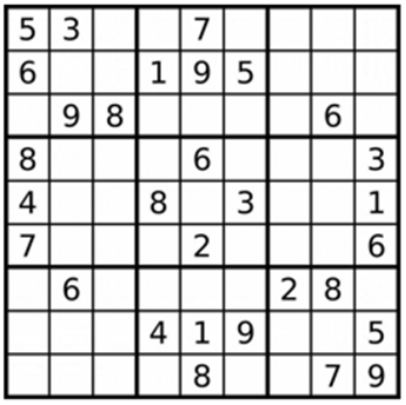
```
void EnhanceSurfaceContrast(SDL_Surface *surface, int C)
{
    float factor = (259 * (C + 255)) / (255 * (259 - C));

    int height = surface->h;
    int width = surface->w;

    for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
            Uint32 pixel = get_pixel(surface, x, y);
            Uint8 r, g, b;
            SDL_GetRGB(pixel, surface->format, &r, &g, &b);
            int newRed = Truncate(factor * (r - 128) + 128);
            int newGreen = Truncate(factor * (g - 128) + 128);
            int newBlue = Truncate(factor * (b  - 128) + 128);
            pixel = SDL_MapRGB(surface->format, newRed, newGreen, newBlue);
            put_pixel(surface, x, y, pixel);
        }
    }
}
```

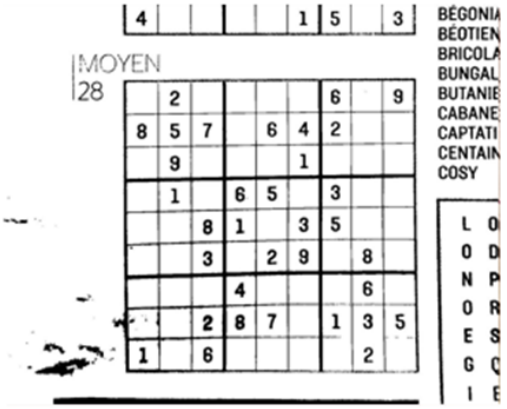
2.1.3.4 Binarization

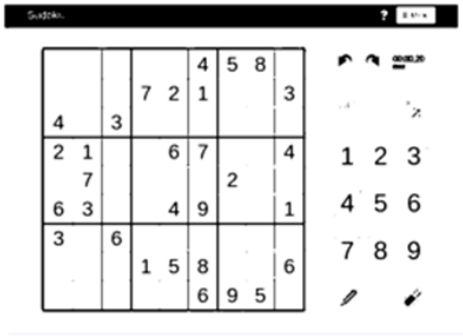
After that, he needed to write a function that turns the pictures to black and white which is also known as binarization. The implementation used to be very similar to grayscale for the first defence, but was heavily modified for the second in order to make it adaptive. The transition was from a global threshold algorithm to a local threshold algorithm. Since it uses a local threshold, a filter size was needed and so he chose a filter size of 3 which will look around the pixel in a 3x3 range. This value was chosen since it is the most efficient and yielded the best results.

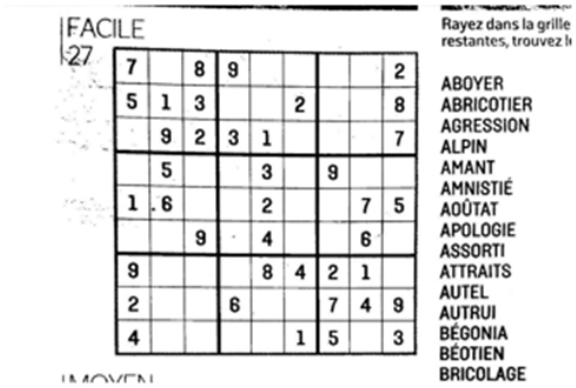


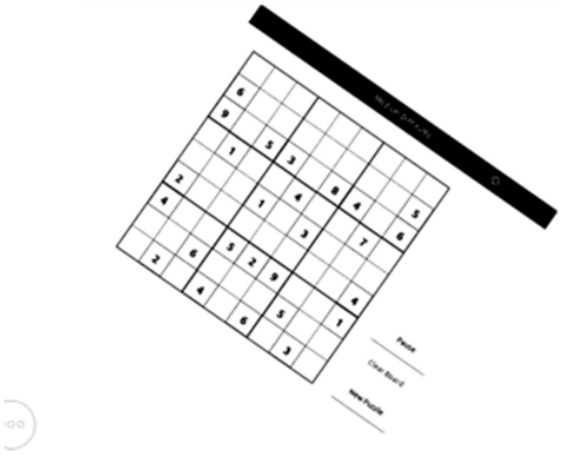
The following pixels are then stored into an array and sorted using the insertion sort algorithm. After testing both the mean (sum of terms / number of terms) and median (middle value of the sorted array), the one which yielded the best results was the median. After getting the median, he then compared it with the global average pixel value of the entire image to determine whether the pixel should be black or white.




MOYEN
28
BÉGONIA
BÉOTIEN
BRICOLA
BUNGAL
BUTANIE
CABANE
CAPTATI
CENTAIN
COSY




FACILE
27
Rayez dans la grille
restantes, trouvez l
ABOYER
ABRICOTIER
AGRESSION
ALPIN
AMANT
AMNISTIÉ
AOÛTAT
APOLOGIE
ASSORTI
ATTRAITS
AUTEL
AUTRUI
BÉGONIA
BÉOTIEN
BRICOLAGE



2.1.3.5 Rotation

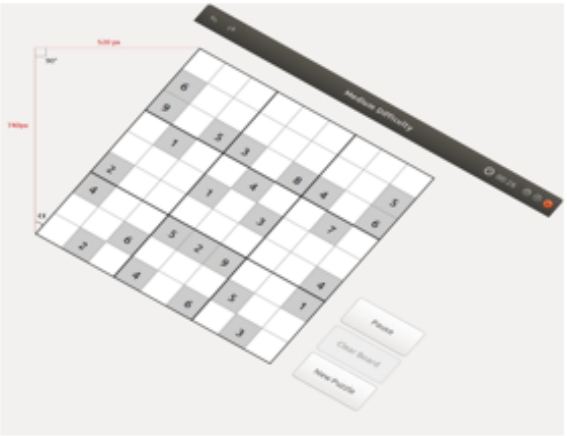
2.1.3.5.1.Manual Rotation

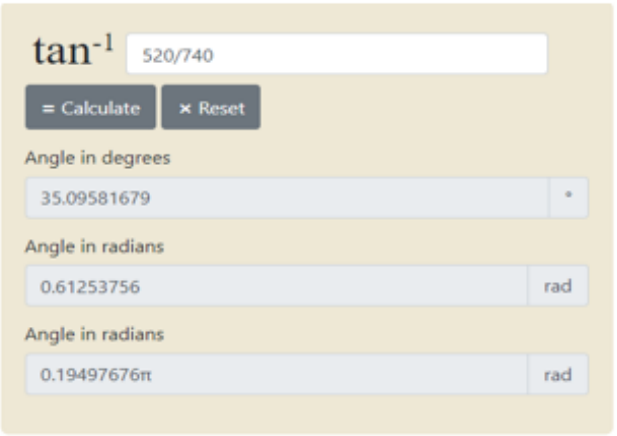
The manual rotation which was done using a rotation matrix. This function uses the “math.h” library. First of all, a new surface is created with the new width and height using SDL_CreateRGBSurface. This function is initialized with R,G,B,A values of 0 for all the pixels which implies a black and non-transparent image. After computing the new width and new height of the new image all that’s left to do is copy the pixels from the old image onto the new sdl_surface that was created. The only downside of this implementation is that the rotation matrix creates a gap between every 4 pixels for the rotated image which will be fixed using noise cancelling.

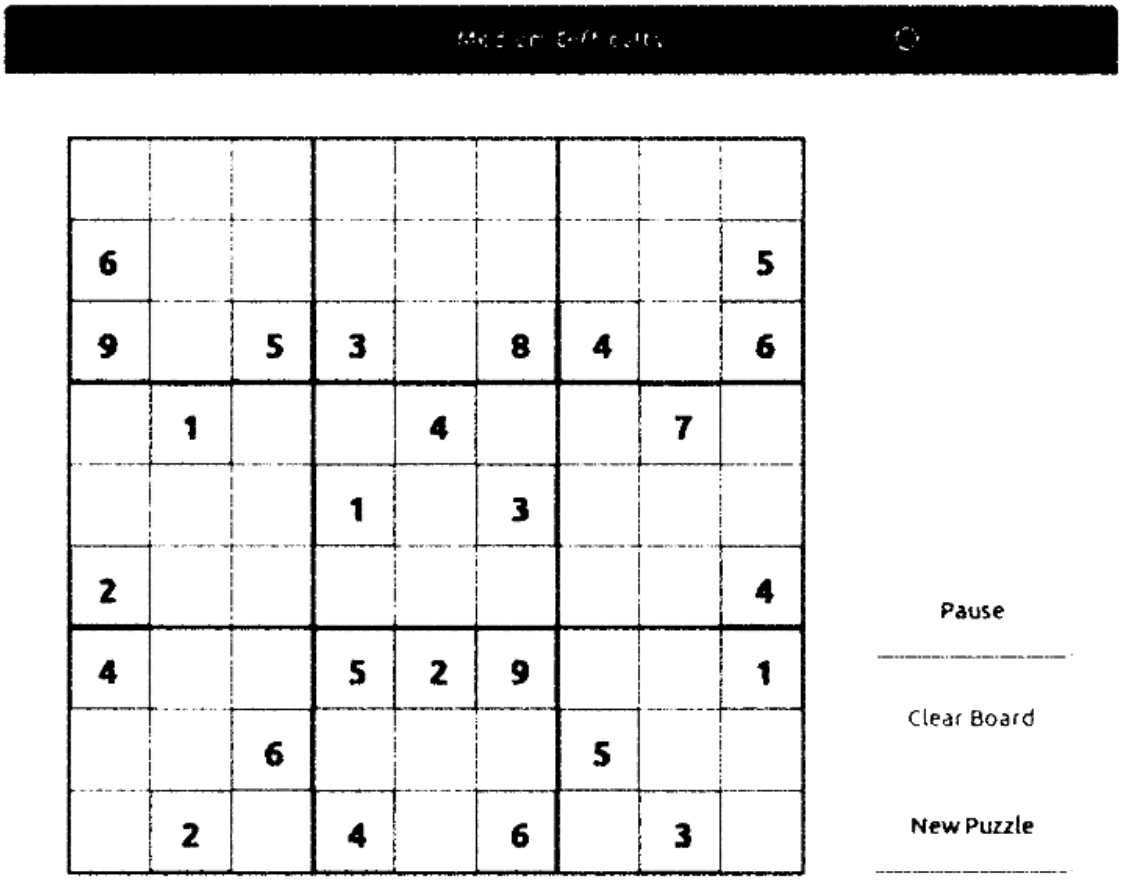
```
SDL_Surface RotateSurface(SDL_Surface *surface, double angleInDegrees)
{
    double angle = angleInDegrees * (M_PI / 180.0);
    double cosine = cos(angle);
    double sine = sin(angle);

    int height = surface->h;
    int width = surface->w;

    int new_height = round(fabs(height * cosine) + fabs(width * sine)) + 1;
    int new_width = round(fabs(width * cosine) + fabs(height * sine)) + 1;

    SDL_Surface* rotated_surface = SDL_CreateRGBSurface
        (0, new_width, new_height, 32, 0, 0, 0, 0);

    Uint32 pixelColor = SDL_MapRGB(rotated_surface->format, 255, 255, 255);
    SDL_FillRect(rotated_surface, NULL, pixelColor);

    int original_center_height = round(((height + 1) / 2) - 1);
    int original_center_width = round(((width + 1) / 2) - 1);

    int new_center_height = round(((new_height + 1) / 2) - 1);
    int new_center_width = round(((new_width + 1) / 2) - 1);

    for (int i = 0; i < height; i++)
    {
        for (int j = 0; j < width; j++)
        {
            int y = height - 1 - i - original_center_height;
            int x = width - 1 - j - original_center_width;

            int new_y = round(-x * sine + y * cosine);
            int new_x = round(x * cosine + y * sine);

            new_y = new_center_height - new_y;
            new_x = new_center_width - new_x;

            if ((0 <= new_x && new_x < new_width) && (0 <= new_y
                && new_y < new_height) && new_x >= 0 && new_y >= 0)
            {
                Uint32 pixel = get_pixel(surface, j, i);
                put_pixel(rotated_surface,new_x, new_y, pixel);
            }
        }
    }
    return *rotated_surface;
}
```

2.1.3.5.2 Automatic Rotation

For the automatic rotation, what was initially considered was creating a histogram using the binarized image and applying calculations based on that. This idea was disregarded after finding out that the grid detection done was already able to detect rotated grid so he already was able to detect the length in pixels of the rotated grid and can easily calculate the angle by using trigonometric formulas.







This result is the finalized image after the rotation and noise cancelling which will be explained below.

2.1.3.6 Noise Cancelling

For noise cancelling, he used local thresholding with a filter size of 3 (3x3 pixels) and based on one of the rgb values, he computed the mean (also known as mean filter). For the edge cases, he supposed the pixel is white to fill in the array. The mean filter is used for very noisy images or for fixing the gaps caused by a rotated image.

```c
SDL_Surface* ApplyMeanFilter(SDL_Surface *surface, int filter_size) // filter size is 3 so 3x3
{
    int height = surface->h;
    int width = surface->w;

    SDL_Surface* filteredImage = SDL_CreateRGBSurface
        (0, width, height, 32, 0, 0, 0, 0);

    Uint32 pixelColor = SDL_MapRGB(filteredImage->format, 255, 255, 255);

    SDL_FillRect(filteredImage, NULL, pixelColor);

    Uint32* window = malloc(filter_size * filter_size * sizeof(Uint32));

    for(int x = 0; x < width; x++)
    {
        for(int y = 0; y < height; y++)
        {
            int varX = -(filter_size - 2);
            int varY = -(filter_size - 2) - 1;

            for(int i = 0; i < filter_size * filter_size; i++)
            {
                if (i != 0 && i % filter_size == 0)
                    varX += 1;

                varY = next_iteration(varY, -(filter_size - 2), filter_size - 2);

                if (x + varX < 0 || x + varX >= width || y + varY < 0 || y + varY >= height)
                    window[i] = pixelColor;
                else
                    window[i] = get_pixel(surface, x + varX, y + varY);
            }

            insertionSort(window, filter_size * filter_size);

            put_pixel(filteredImage, x, y, window[(int)((filter_size * filter_size) / 2)]);
        }
    }

    free(window);

    return filteredImage;
}
```

2.1.4 Grid detection

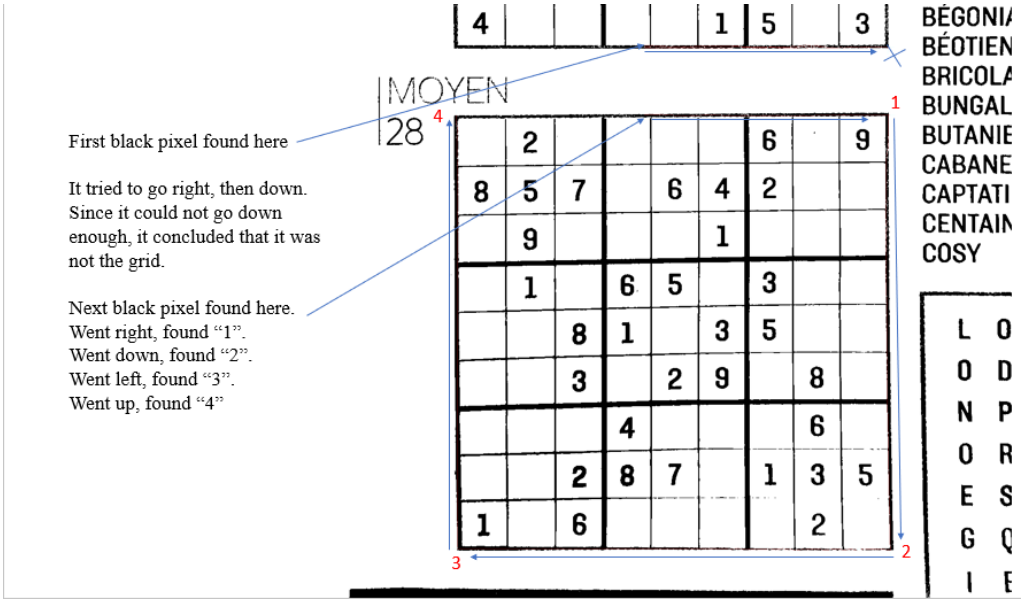
Alexandre started looking into ways to detect a sudoku grid from a black and white picture. He ended up thinking of a method that could detect a sudoku grid in a picture based on the assumption that it is the main part of the picture (at least 3/8 of the picture). He also added a part that would track wherever the function was already called by placing red pixels there. This allowed him to obtain an image showing what the algorithm does specifically. To detect the grid, the algorithm finds the four corners of the sudoku square like so :

Step 1 : It gets the topmost middle pixel of the image.

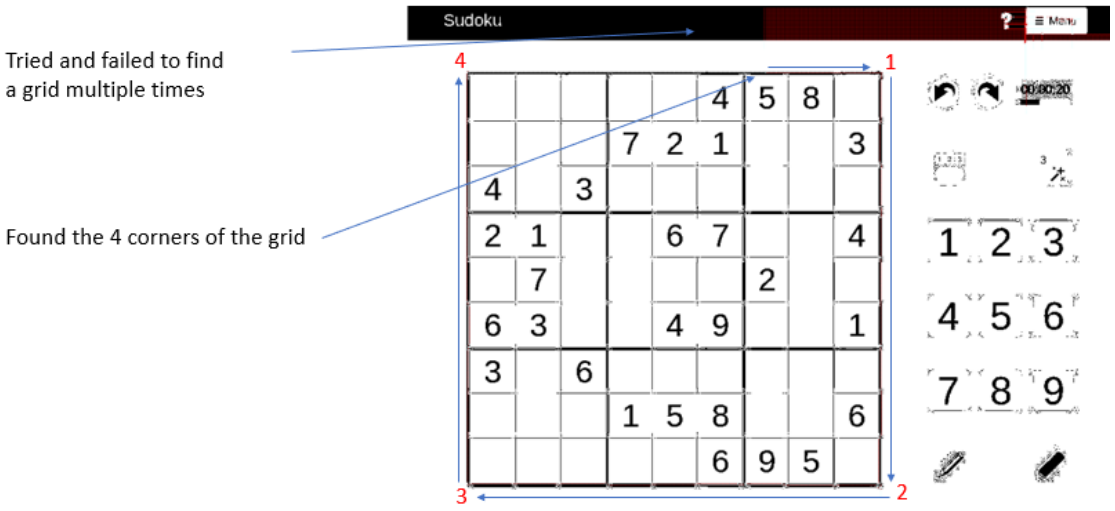
Step 2 : If it is black, it tries to go as far to the right as possible (until it sees white pixels).
This works approximately. It considers that we can keep going if a black pixel is close to our black pixel. That way, it won't mistakenly stop for white pixels that might appear.
Step 3 : If this length is above a certain size (width of picture/8), the algorithm considers that this might be the sudoku. It records the coordinates of this corner and goes to step 4. If not, it goes down two pixels, then goes back to step 1.

Step 4 : Repeat steps 2 and 3 by going downwards and checking if it is at least size of the height of picture/4. If it is, repeat steps 2 and 3 but by going left and checking if it is at least of size width of picture/4. If it is, it does the same but goes up. At that point, we have the four coordinates of the corners. This means that the grid was successfully detected.

Explanation with image :



Explanation 2 :



In order to implement this algorithm, **Alexandre** created a main function which checks the middle pixels of the picture from top to bottom until it finds a grid. Whenever it finds a black pixel, it calls loop1 which will try to go right.

If it can go far enough it will call loop2, then loop3 and then loop4. It then launches a function to get the location of the grid cells determined by the coordinates of the 4 corners.
Between the first and second defence, it was modified so it could detect rotated images as shown as follows:

This was done so we could then get the angle and apply rotation on it.

2.1.5 Detection of grid cells and getting the numbers

Alexandre created a function which takes for parameters the image and the coordinates of the 4 corners of the sudoku. It uses them to find the coordinates of the 4 corners of each of the sudoku cells.
For example, the x coordinate of the first corner of a cell would be
int x=(-(x4+(x3-x4)/9*i2)+(x1+(x2-x1)/9*i2))/9*i+x4+(x3-x4)/9*i2;
With :

- x1, x2, x3, x4 corresponding to the x coordinates of the “1”, “2”, “3”, “4” corners respectively.
- i being the column in which the box is and i2 the row.

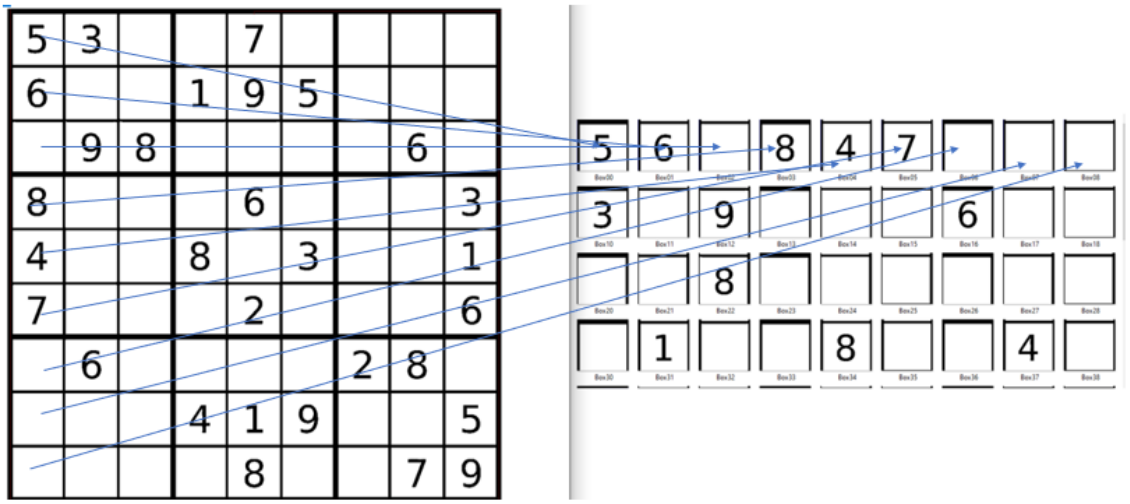
For each cell, the algorithm it applies the following steps :

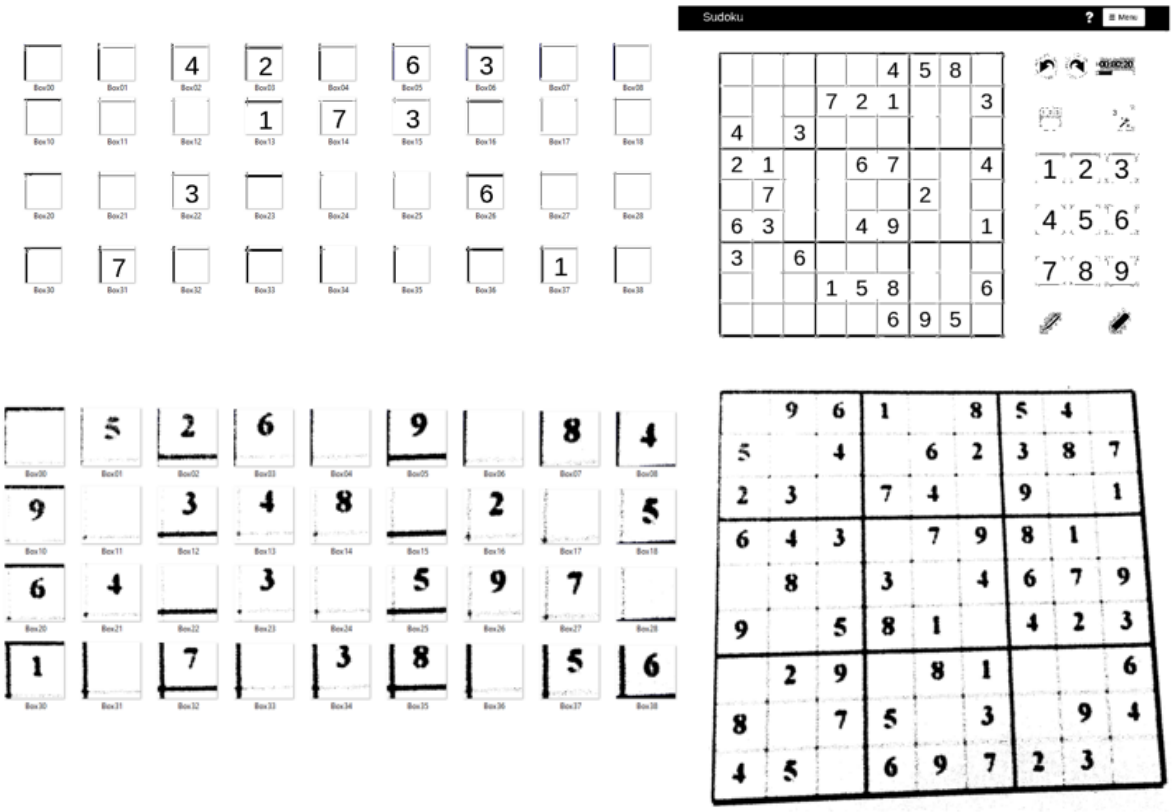
- Retrieval of the cells
- Creates a new image the size of the cell and copies each pixel of the cell in this image,

This image will basically be the image of the number with the grid still surrounding it.
If we save each picture with a different name:
Boxxy where:

- x is the collum (starting from 0)
- y is the row (starting from 0)

The following pictures show the first 4 columns of cells obtained from the sudoku next to them :

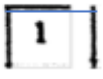




Each of these boxes will get any black lines of the cells and only keep the number that is in it.
For each image it will:

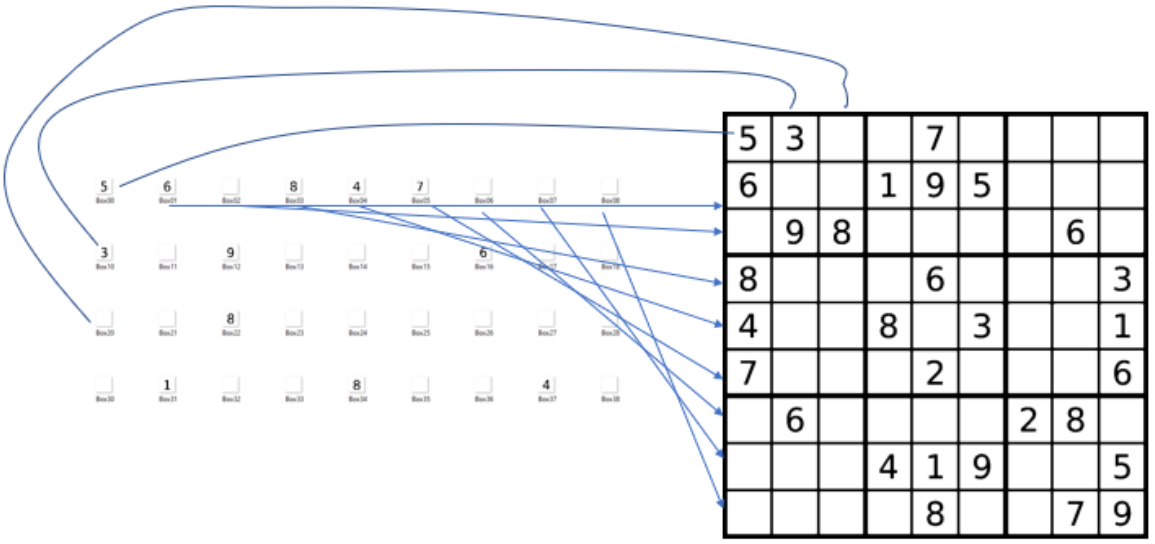
- look at all the pixels in a line (the bleu one in the image) starting at one fourth of the height of the image.

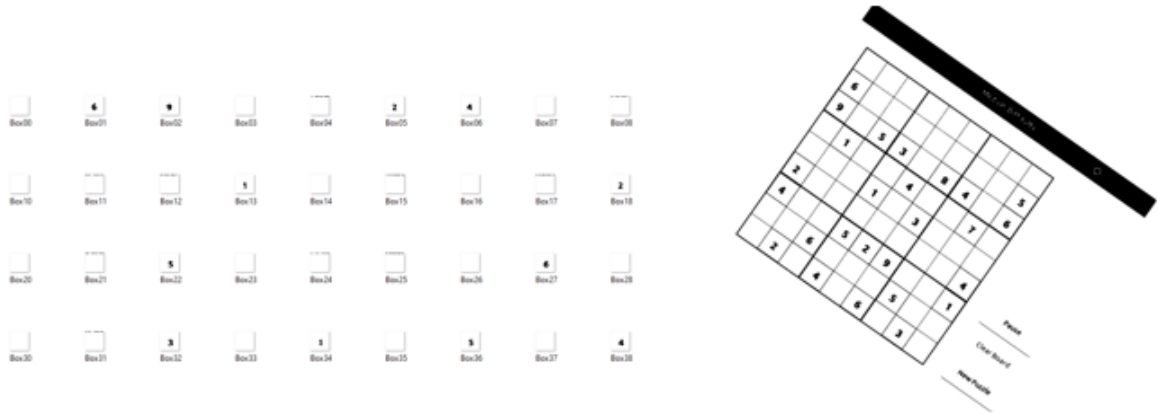
If it finds that there are at least a certain amount of black pixels, it will consider it a line and ignore that part when taking the number from the image.
If it does not, it tries again with a lower y axis and looks at the pixels across this bleu line (approximately)



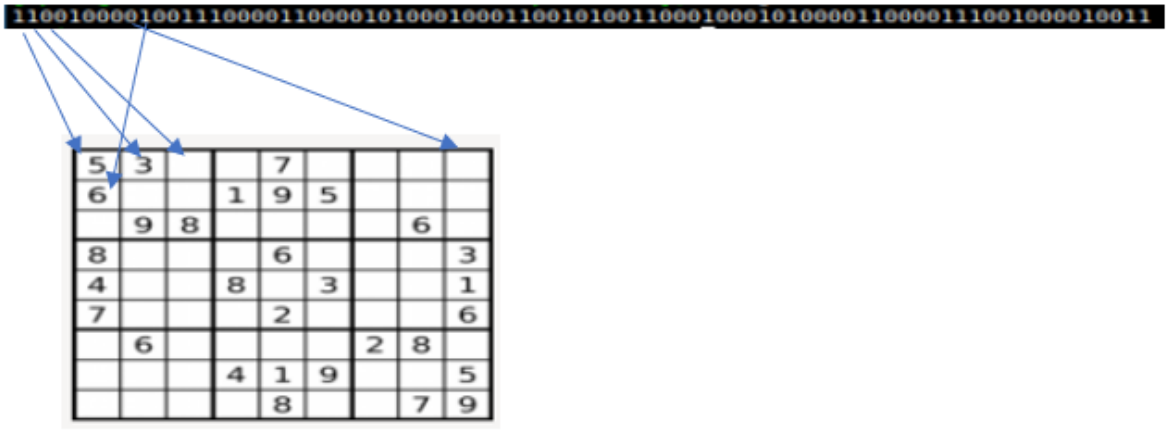
Then across this line where it will find a line of black pixels.
It will then do a a process following the same principle across the three other sides of the image detecting something looking like this:

It will then copy the inside of the image (without the black lines) while transforming its size in order for it to create a 28 by 28 pixel image of the box without the black lines.
It will do so for all the boxes and save them with same naming principle used before:





Furthermore, in order to avoid considering white or almost white boxes as numbers, it will then look at the amount of pixels located in the picture. It will then create a txt file called grid which have1 written in it if the box is a number or 0 otherwise like so:

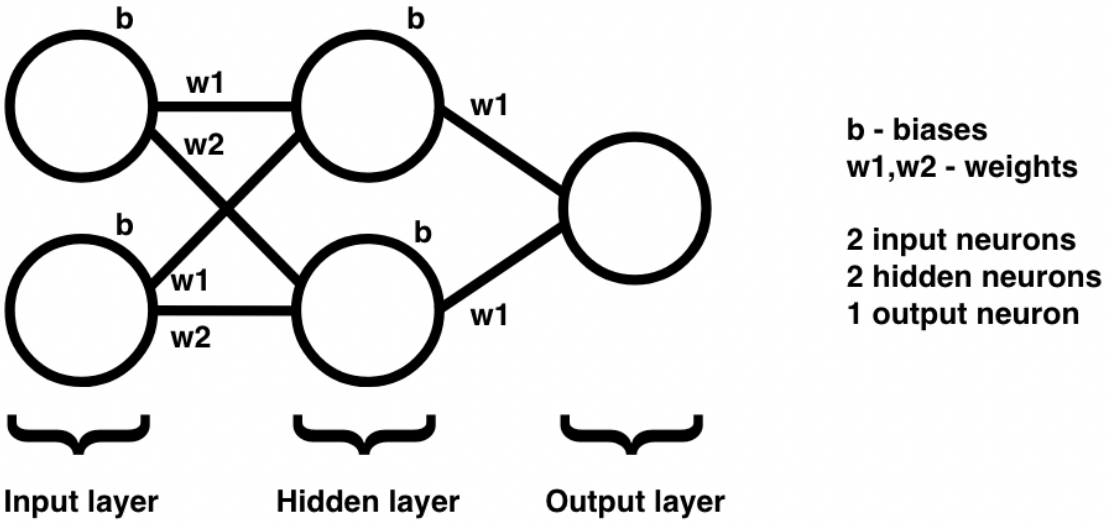


2.1.6 Neural network

2.1.6.1 Structure of the neural network

Barbora was in charge of the neural network. She started building it by creating structs to represent the neuron, layer and the whole network.

For the neural network to be able to recognise the xor function, it was sufficient enough to create a 3 layer network. She thought that building structs from the beginning, even for a neural network with three layers where it is not that necessary, it would be useful later on. The first layer has to input neurons, then the only hidden layer has two neurons and finally the output layer has one output neuron.



2.1.4.2 Training process

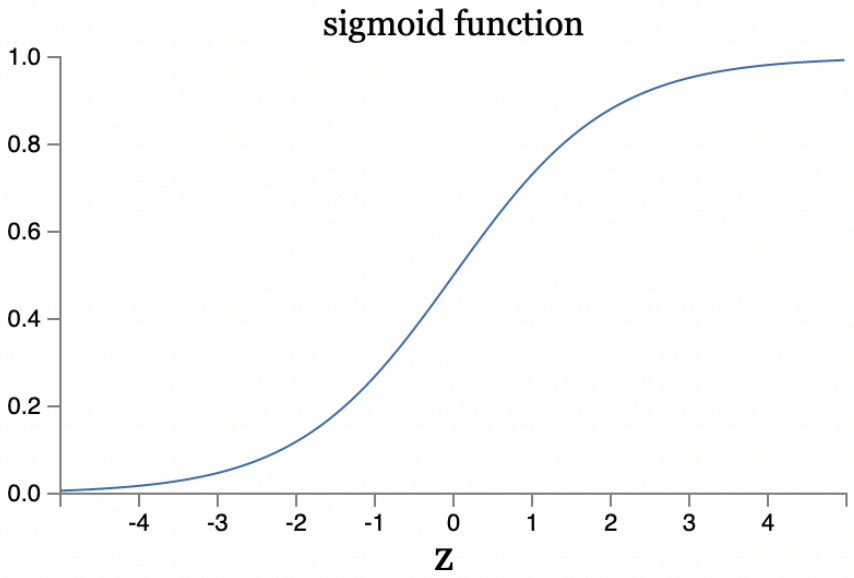
The training process is composed of the front propagation, back propagation and modification of the weights.

The front propagation basically means that we will feed the network some training input and the network will try to guess the output. There is a formula which allows the network to take a guess.

$$\Delta \text{output} \approx \sum_j \frac{\partial \,\text{output}}{\partial w_j} \Delta w_j + \frac{\partial \,\text{output}}{\partial b} \Delta b,$$

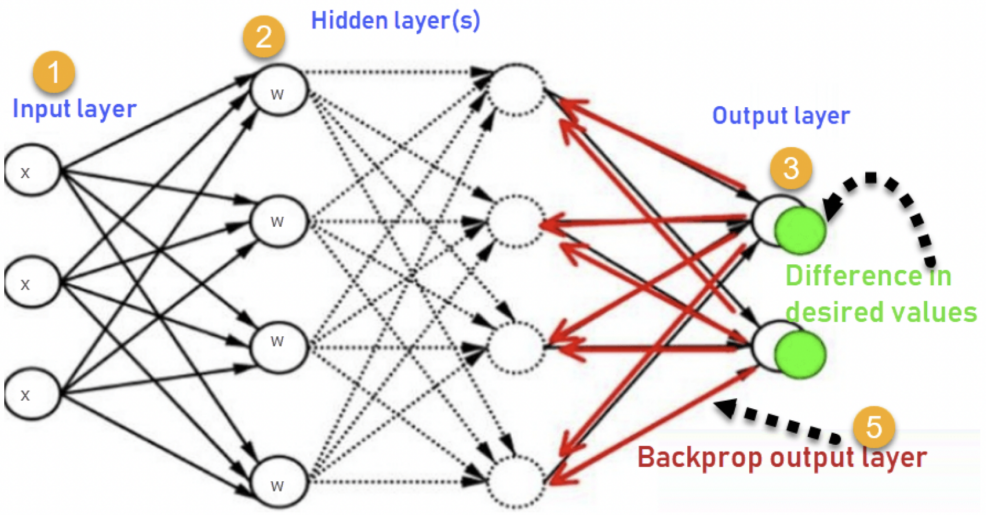
In the picture above, you can see the formula the neural network uses in the front propagation to calculate its guess. It is the sum of all the weights multiplied by the activation value of the neurons and the biases, which is then passed into the sigmoid activation function to get a result between 0 and 1.

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$



This is the sigmoid function which takes a z parameter that represents the sum that was explained above.

The back propagation tries to calculate the cost, meaning how much the neural network's guess was incorrect. Depending on this cost, we will modify the weights of the neural network. The cost is calculated by subtracting the actual output from the expected output, which is then used to modify the weights accordingly.



How Backpropagation Algorithm Works

In the image above you can see a simple explanation of the training process, points 1 to 3 are front propagation, where we feed the network some input and wait for it to calculate the output. Then, points 3 to 5 is the back propagation, where we calculate the error in the outputs and then travel back from the output layer to the hidden layer to adjust the weights such that the error is decreased.

2.1.6.3 Digit recognition

When Barbora started to train the neural network with images of digits, she encountered a few problems with running SDL on her mac. Therefore, at the beginning she started to train the network with the MNIST database of handwritten digits. These images are encoded, so she did not need the SDL library to traverse through them.

THE MNIST DATABASE

of handwritten digits

Yann LeCun, Courant Institute, NYU
Corinna Cortes, Google Labs, New York
Christopher J.C. Burges, Microsoft Research, Redmond

Please refrain from accessing these files from automated scripts with high frequency. Make copies!

The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image.

It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting.

Four files are available on this site:

```
train-images-idx3-ubyte.gz:  training set images (9912422 bytes)
train-labels-idx1-ubyte.gz:  training set labels (28881 bytes)
t10k-images-idx3-ubyte.gz:   test set images (1648877 bytes)
t10k-labels-idx1-ubyte.gz:   test set labels (4542 bytes)
```

All the images from the MNIST database had a size of 28 by 28 pixels and so the images that we would eventually get from our pictures of sudoku, would have to have the same size.

By training with only the mnist database, she has achieved an accuracy rate of 91%, as can be seen on the image below.

```
Epoch 0: TRAINING: Reading image No. 60000 of 60000 images [100%]  Result: Correct=52244  Incorrect= 7756  Accuracy=87.0733%
Epoch 0: TRAINING: Reading image No. 17806 of 60000 images [ 29%]  Result: Correct=14276  Incorrect= 3530  Accuracy=80.1752%
2: TESTING:  Reading image No. 10000 of 10000 images [100%]  Result: Correct= 9164  Incorrect=  836  Accuracy=91.6400%
```

However, when testing the neural network with the images from the sudoku, the success rate was not as high. So she decided to test with computer generated images and she got a 100% success rate.

She has also tried to test with both of these data and the success rate has stayed around 90% and this file with weights is saved in the project repository under the name “handwrit-weights.txt”.

2.1.7 Save and load of weights

Barbora also did the save and load functions of the weights from the already trained neural network. The save function will save the weights and biases on each line. First it saves the bias of a neuron and then all the weights. After that, each layer is separated with a *.

For the load function, she decided to change from getDelim to getLine which is a little more convenient and then since she knows the number of weights and biases, she could easily associate them with their appropriate neurons and layer.

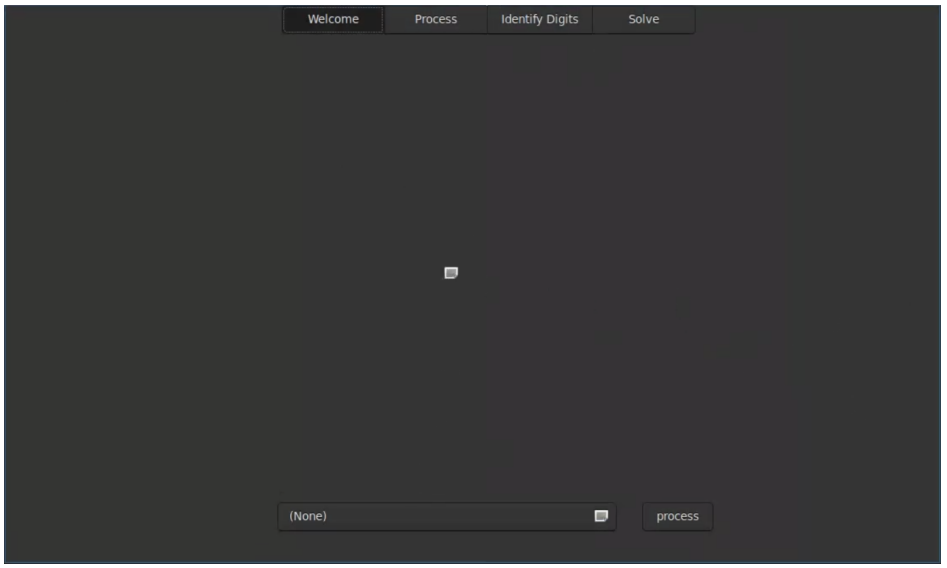
2.1.8 Graphical User Interface

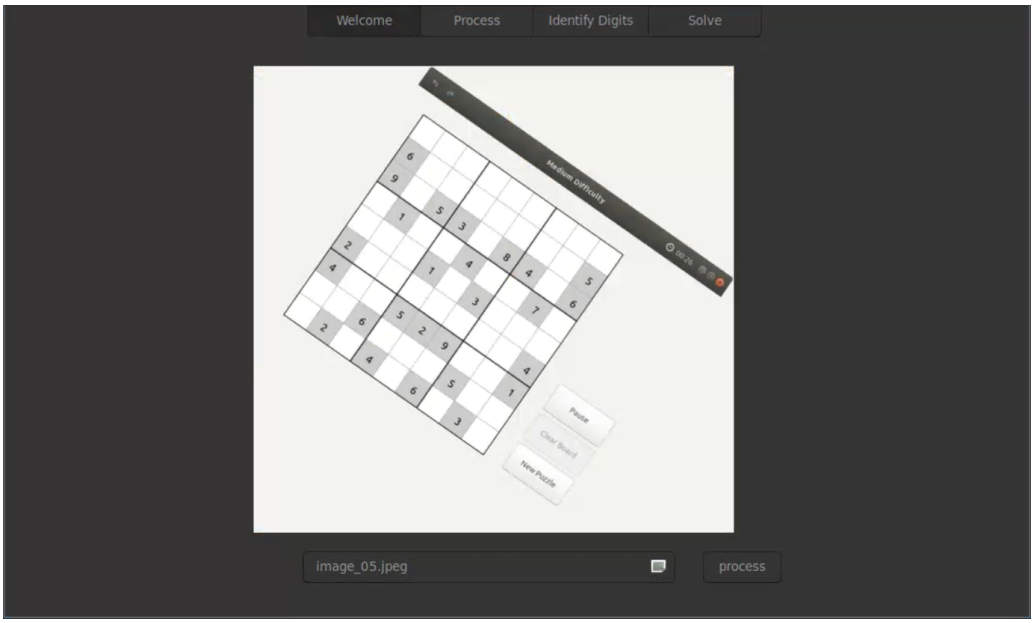
Elsa was in charge of creating the GUI. She designed it using Glade, and it didn't seem as obvious at the very start. She was planning on creating one GtkWindow per page, but after a bit of research she found that working with a GtkStack placed on a single main window would be the most optimal for the final product.

2.1.8.1 GUI Structure

The GUI is composed of a window, on which she placed a GtkFixed and a GtkStack that has 4 pages. Then, she placed a GtkStackSwitcher to be able to navigate between the different pages with ease. Each page has different GtkWidgets (GtkButton, GtkImage) that correspond to its purpose.

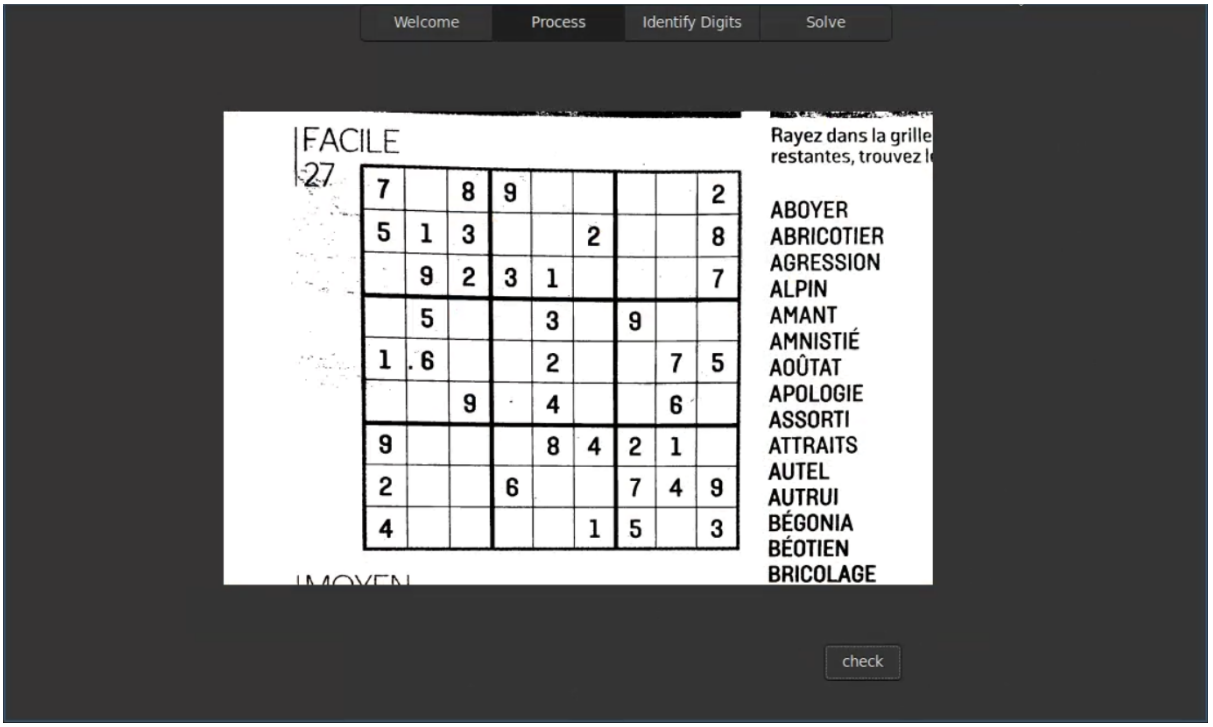
2.1.8.2 Welcome Page

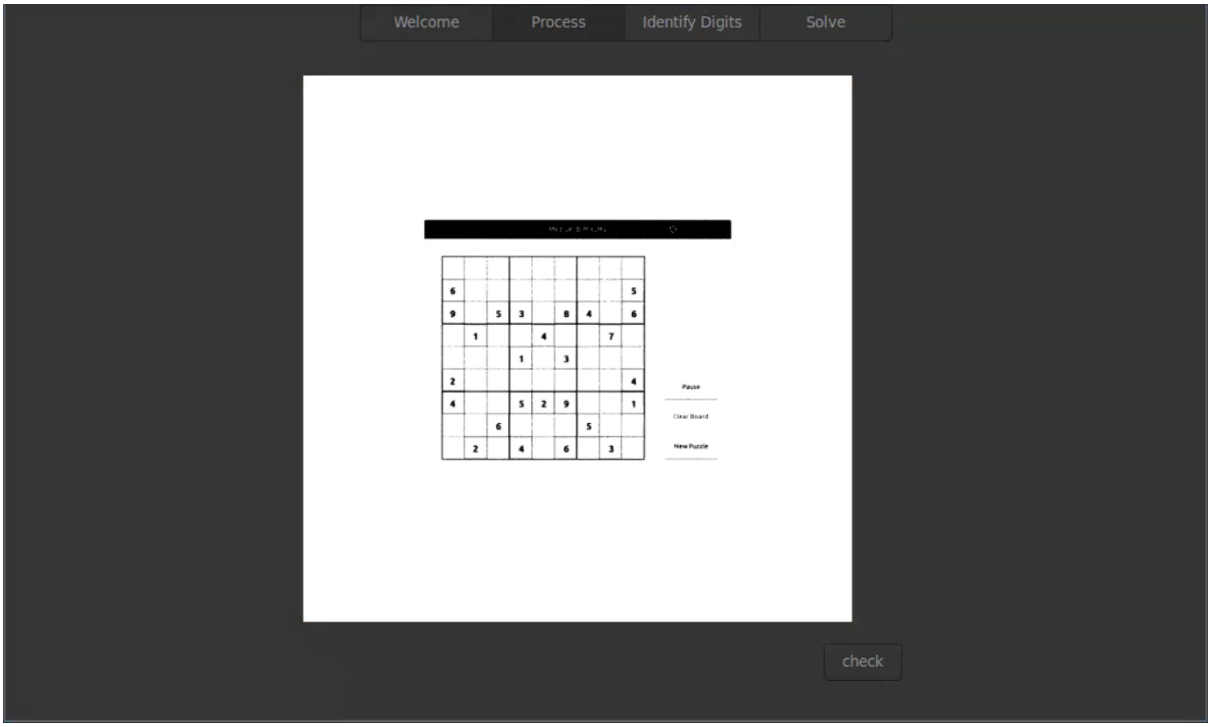




This is the first page the user sees when they open the application. Its main component is the GtkFileChooserButton that allows the user to choose an image. The chosen image will then be displayed on the screen. The ‘process’ button at the bottom right of the page will pass the image through the processing created by **Enrique** and **Alexandre**, and lead the user to the second page of the stack.

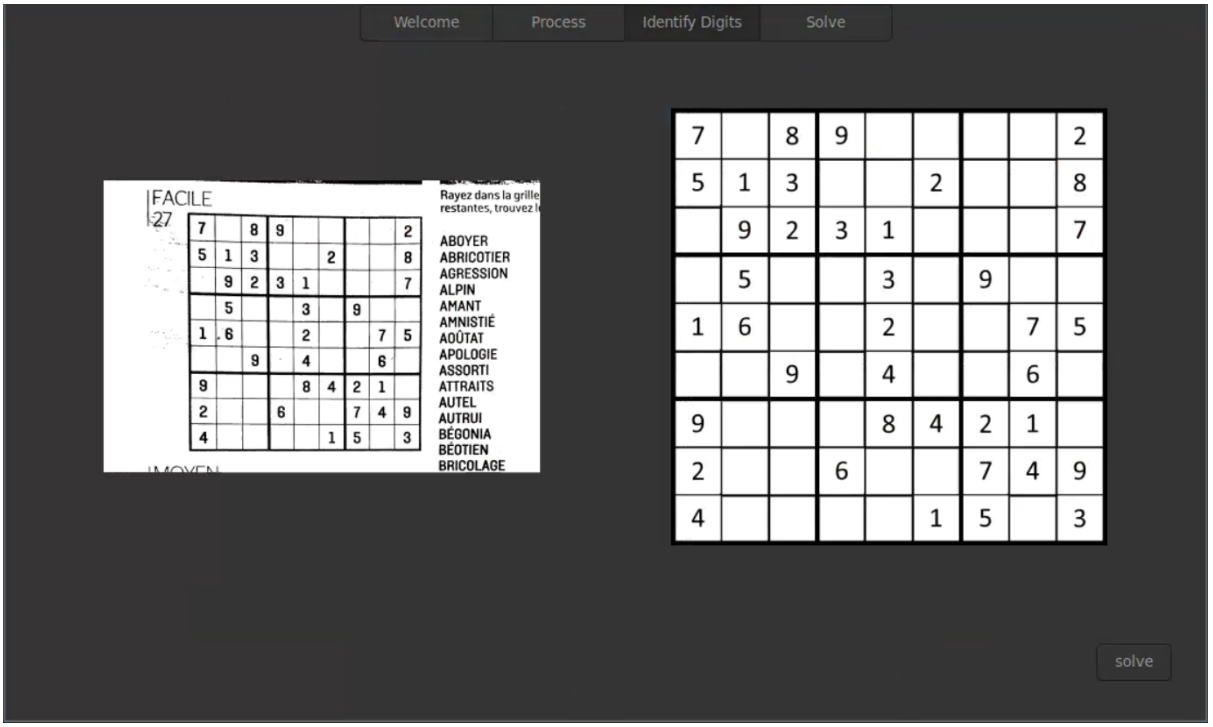
2.1.8.3 Process page

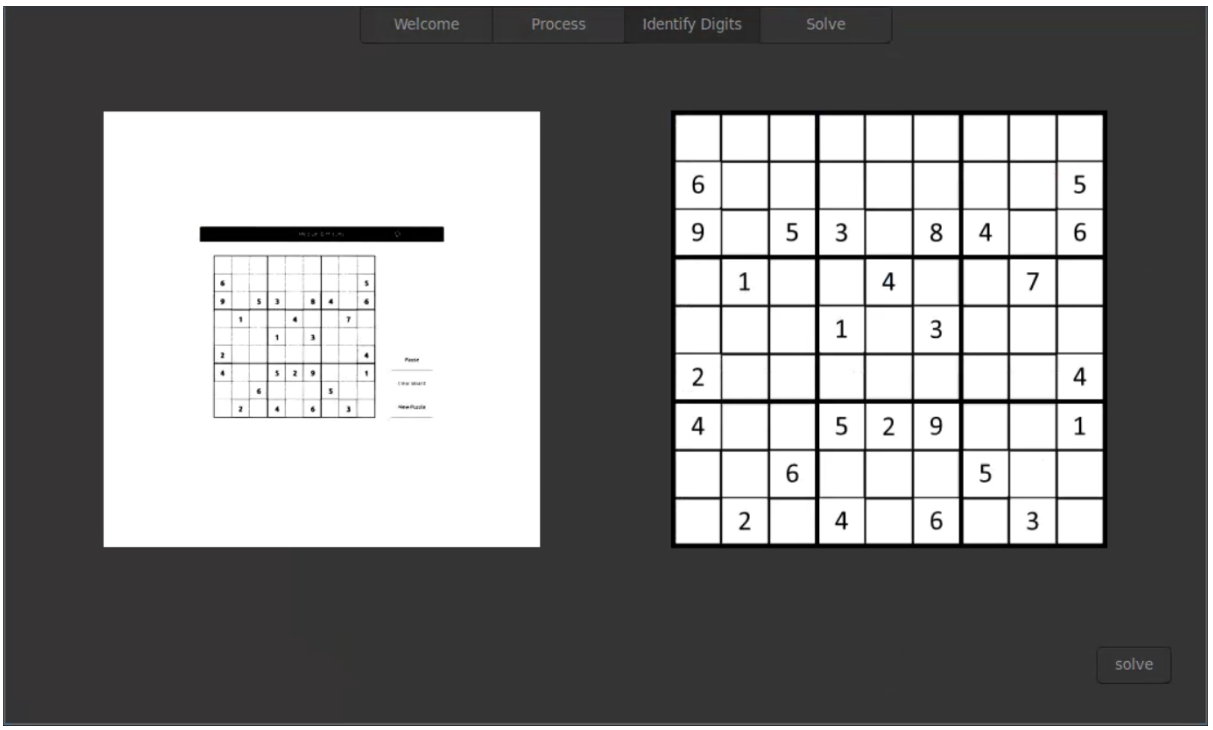




Clicking the ‘process’ button starts the image processing and displays it in the middle of this page. At the bottom, the user can see a ‘check’ button which will lead them to the next page.

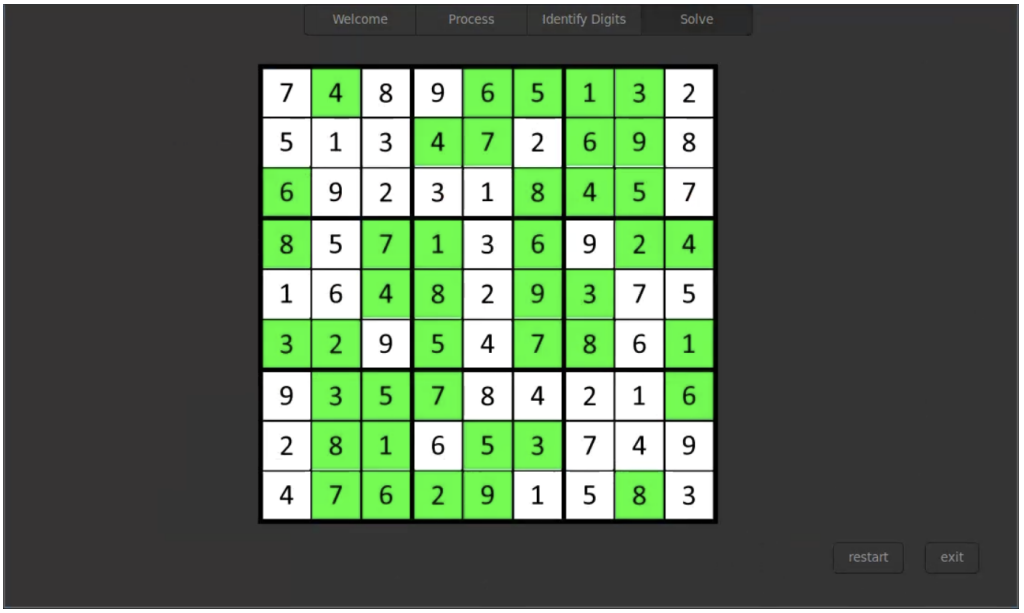
2.1.8.4 Identify Digits page

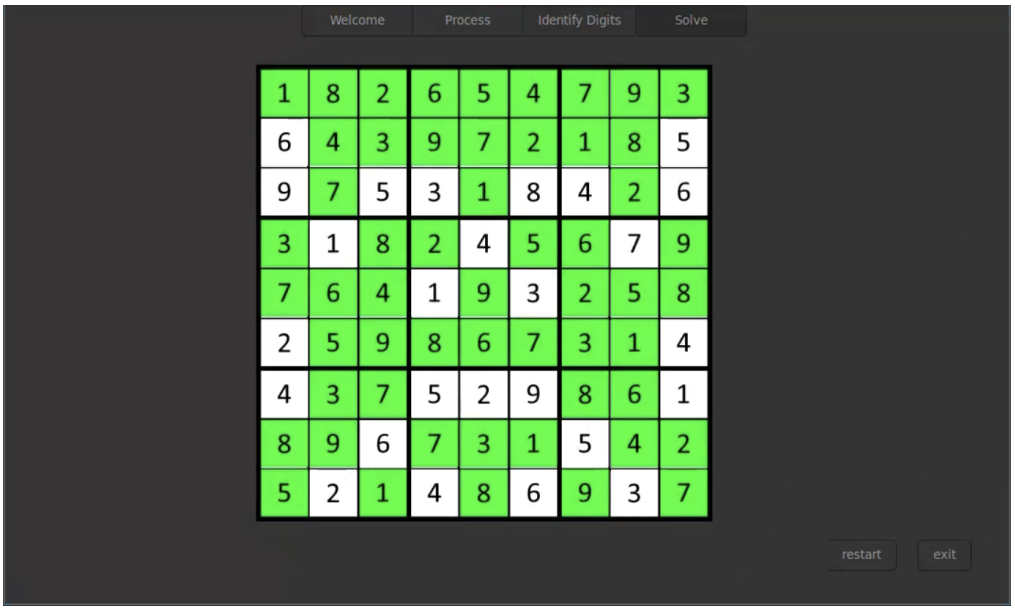




Here, the user will see the result of the splitting and the neural network working together, as well as the creation of a new sudoku grid. In fact, there are two images displayed : the regular processed image from the previous page, and the newly generated sudoku grid containing the digits recognized by the neural network. At the bottom of the page, there is a ‘solve’ button which will lead the user to the final page.

2.1.8.5 Solve page





This page displays the image of the final solved sudoku grid, generated by **Enrique**'s code. At the bottom of the page, the user can find a 'restart' button which will reset all the previous information and take them back to the 'Welcome page'. The 'exit' button exits the application.

2.1.8.6 Back-end

```
typedef struct UI
{
    GtkWindow* window;
    char *filepath;
    gchar* filename;
    GtkButton* solve_button;
    GtkButton* process_button;
    GtkButton* check_button;
    GtkButton* restart_button;
    GtkButton* exit_button;
    GtkFileChooserButton* file_chooser;
    GtkImage* processedImage;
    GtkImage* chosenImage;
    GtkImage* solvedImage;
    GtkImage* oldImage;
    GtkImage* generatedImage;
    GtkStack *stack;
    SDL_Surface *image_surface;
    int** recGrid;
}UI;
```

Creation of the struct UI.

Elsa created a struct UI in which she stored all of the components needed for everything to work seamlessly. She then initialized it in the main function, as well as separately initializing all the instances of its components.

In order to initialize the Gtk objects, she initialized Gtk and created a builder which is created using the 'gui.glade' file she designed.
The objects are then initialized using (gtk_builder_get_object(builder,"object ID"))

```c
int main (int argc, char **argv)
{
    // Initializes GTK.
    gtk_init(&argc, &argv);

    // Constructs a GtkBuilder instance.
    GtkBuilder* builder = gtk_builder_new();

    // Loads the UI description.
    // (Exits if an error occurs.)
    GError* error = NULL;
    if (gtk_builder_add_from_file(builder, "gui.glade", &error) == 0)
    {
        g_printerr("Error loading file: %s\n", error->message);
        g_clear_error(&error);
        return 1;
    }

    g_object_set(gtk_settings_get_default(),"gtk-application-prefer-dark-theme",TRUE,NULL);

    // Gets the widgets.
    GtkWindow* window = GTK_WINDOW(gtk_builder_get_object(builder, "window"));
```

Gtk initialization in main function

2.1.8.6.1 Linking the buttons

The buttons are linked to their respective functions using G_CALLBACK and g_signal_connect.
instance : GtkButton.
detailed_signal : “clicked” for when we click the button.
c_handle : G_CALLBACK(respective on_function).
data : address of the initialized instance of our UI.

```c
#define g_signal_connect (
  instance,
  detailed_signal,
  c_handler,
  data
)
```

Declaration of the g_signal_connect function

2.1.8.6.2 Image display

For the image display, she encountered issues with the image sizing when loading pictures from files so she decided to use pixbuf instead. Here's an example of loading an image to display :

```
gui->filepath = "processed.bmp";
GdkPixbuf *pixbuf = gdk_pixbuf_new_from_file_at_size(gui->filepath, 650, 500, NULL);
gtk_image_set_from_pixbuf(gui->processedImage,pixbuf);
```

Example of image display through pixbuf

3 Realisation of the project

3.1 Task distribution

Task	**Assigned to**	**Substitute**
Image loading and color removal (gray-scale, then black/white)	Enrique	Alexandre
Preprocessing (manual rotation, automatic rotation, noise cancelling, contrast enhancement)	Enrique	Alexandre
Grid Detection	Alexandre	Barbora
Grid cells Detection	Alexandre	Barbora
Image Splitting	Alexandre	Enrique
Character Recognition (the digits, NEURAL NETWORK)	Barbora	Elsa
Grid Resolution	Elsa	Barbora
Grid Reconstruction	Enrique	Elsa
Grid Saving	Elsa	Barbora
Grid Reconstruction	Enrique	Elsa
Graphical user interface	Elsa	Barbora

3.2 Progress

Task	**1st Defence**	**2nd Defence**
Image loading and color removal (gray-scale, then black/white)	100%	100%
Preprocessing (manual rotation, automatic rotation, noise cancelling, contrast enhancement)	25%	100%
Grid Detection	100%	100%
Grid cells Detection	100%	100%
Image Splitting	100%	100%
Character Recognition (the digits, NEURAL NETWORK)	40%	100%
Grid Resolution	100%	100%
Grid Reconstruction	0%	100%
Grid Saving	100%	100%
Graphical user interface	0%	100%

4 Conclusion

Working on this project has been an extremely enriching experience. We are glad that there have been no issues within the group as we have made communication and helping each other our main priority. Even though we encountered an abundance of setbacks along the way, we never gave up and kept on working through them until a solution was found.
We are proud of our work, both, as individuals, and as a group.