Léopold Tran
Thomas Polo
Matis Fardeau
Achille Guerard

OCR

FIRST REPORT

MASTERO

prépa spé

30th October 2022

OCR

Contents

1 Introduction

This report will serve as a summary of the tasks completed since the begining of this project. The difficulties that were faced and the solutions that were adopted will be explored.

As a quick reminder, the goal of this project is to solve a sudoku grid by taking a picture of it.

2 Our team

Our team is called MASTERO and is composed of four highly motivated student willing to produce the best OCR possible. You can read here the resume of each and every one of us :

- Léopold Tran: I have been programming for some time now, but this project has been really challenging. Throughout the first two months of this project I have learned not only a lot of image processing but also on how to divide the work between a team when each part are so intrinsically linked. The most of my energy has been put into research on image processing, OCRs and sudokus. For this defense I was in charge of the color removal and the grid detection.

- Thomas Polo: For this first defense I was in charge of the XOR Neural Network and splitting the grid into individual squares. Programming is something I had picked up in my last year of high school while trying to find what is something I might enjoy enough to make it a career. This project, unlike the one done in S2, has each person's work need to be able to function within their own file as well as others. This has proven to be a challenge as it requires us to be fully aware of our returns and what is needed to run each function as well as make our code readable enough for someone else to use. This has made this project quite challenging in terms of my abilities while coding as well as working in a team.

- Matis Fardeau: For this project I chose to be in charge of the sudoku solver because I liked the challenge of making the best solver possible with all the different methods that already exist. I have been passionate about programming for some time now and the team work is also something I really enjoy. Sharing code, point of view and advices with my team mates was really heartwarming and I really liked this first part of this project.

- Achille Guerard: I first started programming at the beginning of high school, and this was such a relief because I had found, at last, something that I liked studying. Throughout my last year at EPITA I've learnt so much especially during the S2 project. However, regarding the fact that I was in charge of the 3D and 2D modelling I did not code a lot, I'm glad that this project puts me out of my confort zone. I have chosen to be in charge of the rotation of the image for this first defense. This was quite hard but more on that in the dedicated section. It was such a relief to work in collaboration with such passionate people that I know are motivated to produce the ORC possible.

3 Task distribution

Task	Leopold	Matis	Thomas	Achille
Pre-processing	Main		Second	
Rotation				Main
Grid detection	Main			
Neural network		Second	Main	
Sudoku solver		Main		
User Interface		Main		Second
Reports	Second			Main

4 Pre-processing

4.1 Grayscale

The color removal was pretty straightforward, simply turning each pixel into their grayscale with the following formula. ((0.3 * R) + (0.59 * G) + (0.11 * B)).

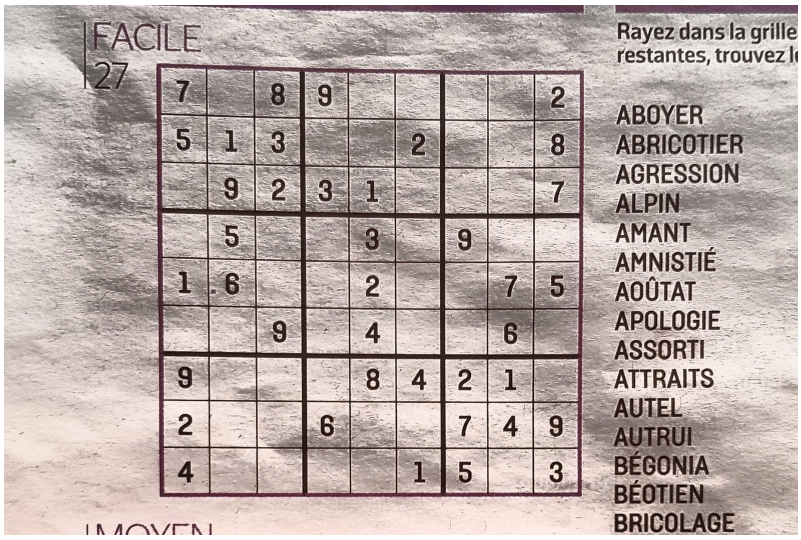

Figure 1: Before the color removal

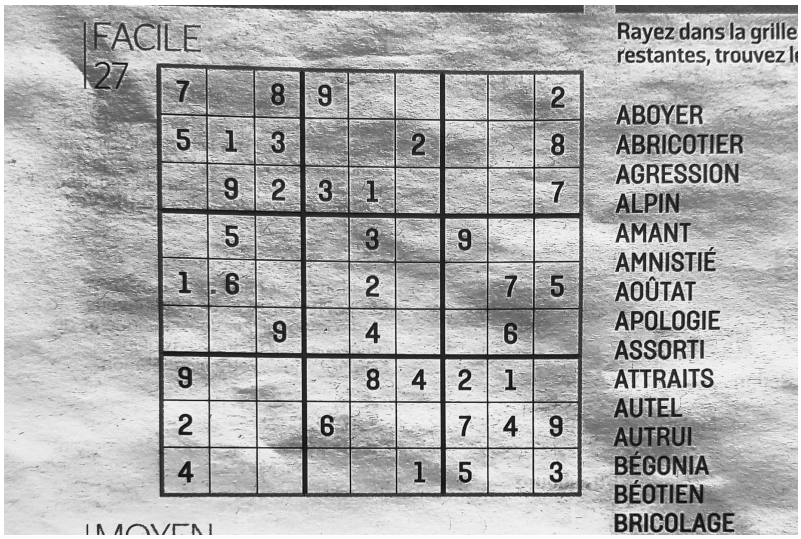

Figure 2: After the color removal

4.2 Rotation

4.2.1 Manual rotation

The first type of rotation that was needed was manual rotation. To do this we use an event loop which matches two keys when pressed : the left and right arrow. Then we rotate our image left and right according to the key that was pressed by the user.

For the rotation we went searched through SDL fuctions since this library is available on school PCs. We found the following function :

```
int SDL_RenderCopyEx(SDL_Renderer * renderer,
                     SDL_Texture * texture,
                     const SDL_Rect * srcrect,
                     const SDL_Rect * dstrect,
                     const double angle,
                     const SDL_Point *center,
                     const SDL_RendererFlip flip);
```

Figure 3: RenderCopyEx function of the SDL library

For the moment we will not use the srcrect and center parameters.

To begin with let us say a few word about our implementation. It is very simple. The event loop waits for a key to be pressed and we have 3 cases :

- if left arrow is pressed : we increment our angle by 0.1f thus rotating clockwise
- if right arrow is pressed : we decrement our andle by 0.1f thus rotating anti-clockwise
- if RETURN is pressed : we output the rotated image as RotDone.png and close the window

The first problem that came to us was the fact that when you render the copy of a texture on the same renderer, it does not remove the first copy. It is pretty straightforward to see the problem here, every time we rotate it generates a whole army of new pixels but it does not remove previous ones. This is an issue in terms of complexity but also in terms of readibility for the grid. If we used this kind of image to perform the grid detection he would certainly fail.

To adress this issue we used the fuction SDL_RenderClear to wipe every remaining pixel of the last grid. This way every time we rotate the grid the last version of it is removed and the new one can replace it.

The second problem that we encountered was the size of the window. When rotating the image if the size of the window does not change some pixels are lost on every side (Top, Bottom, Left and Right) This can be a huge problem sometimes. In fact, in some cases we only lose useless information, for example the bordure of the grid with some random text ect. However in some case we can loose crutial information such that parts of the grid and numbers. This is why we had to adress this issue quickly.

To manage the size of te window we traveled back to our years in secondary school, in geometry class. The initial size of the window is a rectangle, and we all know that every rectacgle is inscribed in a circle, this circle is our rotation. we need to have a window the size of that circle. Finally, we get that the window needs to be a square of size C with :

$$\mathrm{C} = \sqrt{W^2 \times 1.005 + H^2 \times 1.005}$$

Here W and H are respectively the initial width and the initial height of out window. We multiply by a small constant to have a small margin, it'll prove itself useful when grid-detecting.

The last thing that we need to do is to place the image in the center of our newly resized window. To do that we will use the coordinates of the top left corner. Initially these coordinates are (0,0) obvioulsy and we need to change that, otherwise we will loose pixels both the top and left side. This is when the *dstrect parameter of the RenderCopyEx() function becomes useful. We can give in parameter the coordinates of the top left corner of a rectangle to display in a specific location of our window. It only remains now to compute such coordinates. To do this we will again use the math skills we learn in highschool and deduce the following formula :

$$(\mathrm{x,y}) = (\ \mathrm{C} - \mathrm{W}_{2, \frac{C-H}{2})}$$

Even tough the rotation part is completed it still remains one step to pursue. The result of our manual rotation needs to be output. It appears simple and it is in fact there exists just one small problem to adress. we want to output an image and the only fuction that allows this in the SDL library is the following one :

```
int IMG_SavePNG(SDL_Surface *surface, const char *file);
```

As you can see, the fuction takes a surface as a parameter and here is our problem. For the moment we dont have any surface, we are only working with texture and renderers. To solve this issue we just copied the whole array of pixels from the renderer to the surface using the SDL_RenderReadPixels function.

This fuction is very handy because it makes use of pointers. Thus, the surface you put in argument is updated automatically. It is called ReadPixels but in fact it also writes them onto the desired surface. You can see in figure 4 the implementation of our save function.

```
void save(SDL_Renderer* renderer, SDL_Window * window)
{
    int w, h;
    SDL_GetWindowSize(window,&w,&h);
    SDL_Surface* surface = SDL_CreateRGBSurfaceWithFormat(0, w, h, 32, SDL_PIXELFORMAT_RGB888);
    SDL_RenderReadPixels(renderer,NULL,SDL_PIXELFORMAT_RGB888,surface->pixels,surface->pitch);
    IMG_SavePNG(surface,"rot_done.png");
}
```

Figure 4: save function

5 Grid detection

The grid detection however was much more complicated. At first I thought it would be simple, just had to detect the biggest square in the picture, but then I realised that my computer didn't know what a square is, it couldn't even find a line. So I did some more research, stumbbled upon famous algorithms such as Robert-Cross's, Hough's transformation and Sobel's. While a mix of these algorithms would have been great for this project, I took the decision to implement my own version. Not only in order to conserve time and energy but also becaus I wanted to trully understand the issues behind this problem and wanted to try and solve them. I understand that finding shapes accuratly is a large issue in the CS world and in no way am I trying to pretend to have a fix. This is just my way of understanding the importance and mass of the issue.

The main issues that arose where the following :

- How do I find a line?
- When I have all my lines how do I know which ones compose a square ?
- What if I have multiple square ?

5.1 Detecting lines

Let me start by answer the questions from biggest to smallest (not the size of the issue but of the object in space).

If I have multiple squares then, I would take the largest one, therefore, my algorithm assumes that the largest square of each image will always be the sudoku grid. I would have liked to implement a version of the algorithm that checks if each square contains smaller squares, rulling out some possibilities, and then taking the largest one but time ran out.

5.2 Detecting the grid

Now, if I have many lines how do I know which one of them composes a square ? In order to solve this issue I've stored the width and height of each line and know if it is a vertical line or a horizontal line. Then I compare each coordinate with the others, and if the start of a line matches the start or end of another line then it mean that we have an intersection. We only compare horizontal with vertical lines obviously. If you have four distinct intersections but all linkning to another intersection, then you've got a square.

An issue I ran into was that my coordintates where too precise, so sometimes it would not recognize an intersection because it was off by a few pixels, invisible to the human eye but not to the computer. Therefore I know check in a radius that is computed accordingly to the size of the image, if two points are each others radius, then we assume they intersect.

Our main issue was detecting lines, at first glance you would think that you just go throught the image and if it is closer to black than white successivly on multiple pixels then you have a line. However the issue is much more complex than that. What if the square isn't straight, then your line isn't straight either. What if there is a smugg on a pixel cancelling out a perfectly fine line, how do you make the difference between the line of a sudoku grid and the line of a 1. Most importantly, if a picture is taken by hand then the paper might be slightly curved, meaning that your lines will be slightly curved. Cancelling out a perfectly fine line.

Let us answer each issue one by one. Unfortunatly due to a lack of time each idea has not been implemented to the best of my ability.

In order to solve the issue of square not being straight in the image we scan the image rotating it each time until we find a square. If it did a whole circle without finding anything we stop. I am forced to admit that this problem has been solved by brute force.

If a line is cancelled out due to a small white pixel in the middle of the line, then we check a few pixels further along that line to check if the line actually continues, if it does then we start again at that point assuming that it was just an error. If however there is still a white pixel then we assume that the line has stopped. This part of the algorithm relies on some part of luck but the probability of it happening many times are low.

6 Cells detection

6.1 Grid splitting

The grid_split function is very simple in concept all it does is grab the initial image that is provided when the main is run (here we assume that the png of the grid will always be perfect) and one by one creates a PNG file containing the result named "cell_{number of the cell}.png". To achieve this first needed to be able to split each square of the png into quadrants. To do that I grabbed the height and width of the image file and divided it by 9 as a sudoku grid is only 9x9 squares long and wide.

```
SDL_Surface *image = IMG_Load(filename);
int height = image->h;
int width = image->w;
int cell_height = height/9;
int cell_width = width/9;
```

Figure 5: Cell width and height

Once we have that down using the SDL_image library we are able to use the SDL_Rect structure that from an image takes a determined area using the origin coordinates and the size of image (height and width) so for example if we wanted to take the first square of the sudoku grid we would have the coordinates as $x = 0$ and $y = 0$ and the width and height as cell_width and cell_height respectively. Now that we have our square, we can use the SDL_BlitSurface function which grabs a SDL_Rect and an empty surface that we create and pastes the square onto the surface, this now allows us to save the square into a png. This will now be done for each and every square by increasing the coordinates of x and y by the cell_height and cell_width when needed.

```
for (int i = 0; i < 9; i++){
    for (int j = 0; j < 9; j++){
        SDL_Rect rect = {x,y,cell_width,cell_height};
        SDL_Surface *cell = SDL_CreateRGBSurface(0,cell_width,cell_height,32,0,0,0,0);
        SDL_BlitSurface(image,&rect,cell,NULL);
        char *cell_name = malloc(20);
        sprintf(cell_name,"cell_%d.png",n);
        IMG_SavePNG(cell,cell_name);
        x += cell_width;
        n++;
    }
    x = 0;
    y += cell_height;
}
```

Figure 6: The for loop explained above

6.2 Issues encountered

For this part of the code, I did not have real any big issue with the making of the code in itself as it is not something very complicated to understand and make. This was also quite easy to make thanks to our previous experience with the SDL and SDL_image libraries in a previous practical work. This time the biggest issue I had with the split_grid was with the Makefile. The Makefile caused me a lot of issues as I could not manage to add the SDL libraries to it when I compiled this caused for some structures to not have a reference and thus make me unable to test my code. At first, I hadn't realized that the Makefile was the issue which led me down a rabbit hole that wasted more of my time then it helped. I only managed to realize my issue when revisiting the practical's subject file where in the given Makefile it showed how to properly compile the SDL package and library. This really showed me that my issues and lack of knowledge on Makefiles held me back and thus caused unnecessary concern with my code.

6.3 What is next?

Having finished the split_grid all that was left to do was to save the result of the function into a separate file to avoid clutter. To do this I have made for now a temporary fix which saves the images into a folder called output, but this folder must exist in the directory before launching the code otherwise nothing is saved. For the next defense I should fix this issue and make it so that the images don't need to have an existing folder to be saved but instead have a folder be made with the format: "{input_name}_split_output". This would make it easier when manipulating multiple grids to be solved at the same time which will be needed for when we will have our neural network train.

7 Neural network

7.1 The XOR neural network

First, I needed to know how many nodes I would need for each layer:

- Two input nodes as a xor compares only 2 values, these values are a combination of True and False which will be then computed in the hidden layer of the network and later output
- For the hidden inner layer since we only needed to do one comparison, we only needed to have a single internal layer and we will have 2 internal nodes
- Finally on the output layer we only need 1 node as the result of a xor is only a single Boolean

Now before making any of the neural network, we need to make some helper functions. Firstly, we need the sigmoid function and its derivative.

```c
//Sigmoid function and its derivative
double sigmoid(double x) {
    return 1 / (1 + exp(-x));
}
double sigmoid_derivative(double x) {
    return x * (1 - x);
}
```

Figure 7: Sigmoid function and its derivative

We will also need to have a function to initialize the weights and the biases of the neural network (the weights and biases are values et between 0.0 and 1.0).

Now that we have our helper functions setup and an idea of what our neural network will look like we can start coding the network. In this neural network we will be using the Stochastic Gradient Descent which updates the weights based on a single pair of inputs/expected outputs. Although due to using this approach it was also necessary to have the training inputs randomized to maximize the chances of the neural network to converge on a correct solution when it is done.

A	B	A **XOR** B
0	0	0
0	1	1
1	0	1
1	1	0

Figure 8: Training input for the xor

For our main loop we will be iterating over the training set for a set number of epochs (10,000 in our case), we will also need to randomize the order of our training sets to make our network as efficient in its learning as possible. But before any of that we will need to initialize our biases and weights in and store them. This is essential as without them your neural network will not learn anything and when testing it you will not get a result close to what to expect. This was one of my biggest issues with the neural network as at first, I didn't really understand the use of the biases and weights which led me to never even initializing them and thus debugging a working loop with just faulty weights and biases.

```
$./neural_network_xor
Training complete!
Testing network:
Input: 0.000000 0.000000 Output: 0.015790
Input: 1.000000 0.000000 Output: 0.673846
Input: 0.000000 1.000000 Output: 0.673846
Input: 1.000000 1.000000 Output: 0.674756
$
```

Figure 9: Xor result without initializing weights and biases

The first function part inside the loop computes the activation of the inner layer (note we don't need any code for the input layer as it is just inputting the training sets), here we will run a single formula on the hidden layer bias which uses the sigmoid function.

$$hiddenLayer_j = sigmoid\left(hiddenLayerBias_j + \sum_{1}^{k} training\_input_k * hiddenWeight_k^j\right)$$

Now, we will do the same for the output layer activation using this function:

$$outputLayer_j = sigmoid\left(outputLayerBias_j + \sum_{1}^{k} hiddenLayer_k * outputWeight_k^j\right)$$

After both are activated, we will need to now calculate the change that has appeared in the networks weights which will move the network towards reducing the error of the output that the network just computed. For this step we first calculate the change in weights (delta_output) by calculating the derivative of the error and multiplying it by the derivative of the output of that node (This is called the Mean Squared Error).

```
// Compute change in output weights
double delta_output[num_outputs];
for (int j = 0; j < num_outputs; j++){
    double error = training_outputs[k][j] - output_layer[j];
    delta_output[j] = error * sigmoid_derivative(output_layer[j]);
}
```

Figure 10: delta_output loop

We will do something similar for the hidden layer with the exception that the error calculation for a given hidden node is the sum of the error across all output nodes with the appropriate weight applied to it.

```
//Compute change in hidden weights
double delta_hidden[num_hidden];
for (int j = 0; j < num_hidden; j++){
    double error = 0.0f;
    for (int p = 0; p < num_outputs; p++){
        error += delta_output[p] * output_layer_weights[j][p];
    }
    delta_hidden[j] = error * sigmoid_derivative(hidden_layer[j]);
}
```

Figure 11: delta_hidden loop

We will then finally update the weights in both the hidden and output layer this will eventually cause the neural network to minimize its mistakes and give a relatively accurate answer. For example, in this xor the results that are given when testing after the training stage have an almost 95% accuracy. This can be changed by either upping or lowering the number of epochs thus making the neural network respectively more or less accurate.

```
$./neural_network_xor
Training complete!
Testing network:
Input: 0.000000 0.000000 Output: 0.054012
Input: 1.000000 0.000000 Output: 0.945577
Input: 0.000000 1.000000 Output: 0.945663
Input: 1.000000 1.000000 Output: 0.055685
$
```

Figure 12: Xor results, 95% accurate

7.2 Issues encountered

While making the xor I had a wide range of issues with the most problematic being the forgetting of the initialization of the weights and biases as previously explained. However even if it was the most time consuming issue I had with the xor, I struggled harder with understanding the xor. At first being the first ever time being faced with making a neural network I had a hard time understanding how and what a neural network was supposed to do. Using the provided explanations as well as external help from other websites, videos, and people I ended up being left more confused and lost then initially as now I understood conceptually how it worked but had no idea where to start and thus trying to go ahead and making it on my own which, in hindsight was very foolish of me. At the end I managed to finish making the xor which helped me realize where I lacked in my abilities as a programmer, as the previous project we could see visually when the game wasn't working as intended and maybe why the issues occurred working on a neural network forced me to be more theoretical in my approach to debugging as I could no longer see where the issues were stemming visually. This helped me realize how much I need to learn how to properly debug and how to properly organize my code for it to be clear to myself as well as to others.

7.3 What is next ?

After finishing the neural network, I had to find a way to save the weights and biases to not have to train the network each time I wanted to run it. To achieve that I made a sort of pseudo code that I just need to adapt and to implement in the code I have right now to eventually be able to save the weights and biases into a separate file which will be useful later for when we use a neural network to identify numbers which will most probably require the neural network to run for longer then just a second which will then gave a use to saving the weights and biases. I have yet to code this part due to running out of time and needing to finish the separating of the grid image into individual squares.

8 Sudoku solver

8.1 The terminal interface

The compiled program takes a single input through the terminal, the input file. This input file is a text file containing a sudoku with the exact same format as presented in the book of specifications. While it is possible to add more formats, considering this program is supposed to be a part of the whole sudoku image solver and not to be used as a standalone, it is much simpler and clearer to only use the format shown.

Once the sudoku has been resolved, the output is written and stored on a separate file named after the original with the suffix ".result", this file keeps the same format by replacing the dots of the original with their respective numbers. Should the input sudoku be already solved or be unsolvable, the result file will be a copy of the original.

The solver.c file can be compiled with gcc like any other .c file although it uses some extra libraries like errx.h which might cause some issues on other systems but it does work on the school-provided environment.

8.2 The program

The execution program itself has multiple steps internally. To begin, the program opens the input file via the standard c file I/O, we proceed by reading a line then removing the whitespaces and the newlines as well as replacing the dots with zeros, this processed line is then added to the board array which contains the cells of the sudoku. Once all the lines have been read, the file is closed and the solve method is called on the board array.

The method first determines if there are any zeros in the board: if there are none, that means that the board is already solved, but if there are zeros, it calls the recursive part of the program.

The recursive method works using the backtracking algorithm on each cell of the board:

1. Have we passed the final cell? If yes, then the board is solved and return true
2. Is the cell empty? If not, skip it and move to the next cell
3. If the cell is empty, call the safelist method which will create a list of the possible numbers that can be in that cell by looking at the numbers that do not appear in the row, column, or square of the cell
4. Then, take the first number in the safelist and put it in the cell before moving on to the next cell
5. If a cell has no available number to choose from or has reached the end of its safelist, reset its number to zero and backtrack to the previous cell by returning false
6. In the previous cell, if the returned value is true, then the board is solved so continue the return true chain, else, replace its number with the next available one

8.3 Our implemetation

The solver function is divided into 2 main functions rsolve and safelist, where safelist builds a list of all possible options for numbers and rsolve tests the outcome of each grid with each number as a possibility.

For safelist, the function starts off by checking all the numbers on the same line and column as the cell which was called with the function and then in the list at position 0 to 8 it changes the value from 0 to 1 if the number can be inserted.

```
int p = (cell / 9) * 9; //start position
for (int i = p; i < p + 9; i++) //check line
{
    if (i == cell || board[i] == 0) //is cell empty?
        continue;
    v[board[i] - 1] = 1; //set equivalent position in v
}

p = cell % 9;
for (int i = p; i < p + 73; i += 9) //check colum
{
    if (i == cell || board[i] == 0)
        continue;
    v[board[i] - 1] = 1;
}
```

Figure 13: Part of safelist used to check line and column

Now that we have the possible values for the line and the columns, we must do the same thing with the square in which the number is located. If the values can be put in the square, then the value at the index – 1 will have its value changed from 0 to 1 much like for the line and columns.

```
p = ((cell / 27) * 27)  + (((cell % 9) / 3) * 3); //upp
for (int i = 0; i < 3; i++) //line loop
{
    for (int j = 0; j < 3; j++) //colum loop
    {
        if (p + j == cell || board[p+j] == 0)
            continue;
        v[board[p+j]-1] = 1;
    }
    p += 9; //next line
}
```

Figure 14: Part of the safelist used to check the square

Now that we have the list of all the possibilities for the cell, we can now exit the safelist functions and go back to the rsolve.

For the rsolve we get the list given by safelist and then try to solve the board for every possible combination till we get one that works.

```
if (board[cell] == 0) //is the cell empty?
{
    int c = 0;                                  //
    int v[] = {0, 0, 0, 0, 0, 0, 0, 0, 0}; //
    safelist(board, cell, v);                   //

    for (int i = 0; i < 9; i++)                 //
    {
        if (v[i] == 1)                      //is n
            continue;
        board[cell] = i+1;                  //put
        c = rsolve(board, cell+1);          //move
        if (c)                              //has
            break;
    }

    if (c)                                      //
        return 1;                           //repo
    else
    {
        board[cell] = 0;                    //rese
        return 0;                           //back
    }
}
```

Figure 15: The resolve function

Now that we have the solver ready we just need to have the main function open the file with the grid to finish and at the end create a new file where we deposit the results of the grid.

8.4 Comments about complexity

Compared to the other parts of the project, the sudoku was probably one of the easiest. Nevertheless, it is still an important part of the project meaning that we need to take care to create an efficient and fast program.

we chose to use a recursive method for the solver method as each recursion state is temporarily stored, including the variables used to track which numbers can be used, this cannot be achieved in a simple manner with an iterative method, not without creating a large list or a large number of variables.

The safelist created for each cell is used to check whether the board is valid for x number, by using this method, we avoid having to verify each of the potential numbers of cell individual, to put it simply, we find all the possible numbers in one scan of the board making the program multiple times faster.

9 Global progression

Task	Current progression	Status
Image loading	100%	Done
Color removal	90%	To finish
Pre-processing	50%	To finish
Rotation	75%	To finish
Grid detection	100%	Done
Grid detection of cells	100%	Done
Retrieval of the digits present in the cells	0%	To do
Neural network	60%	To finish
Sudoku solver	100%	Done
User Interface	15%	To Finish

10 Conclusion

To conclude we would like to say that we are very proud of our advancement. We managed to handle most of the pre processing, including the grascale, the binarization and the manual rotation.We also split the grid sucessfully into 81 different cells. Furthermore we implemented our first neural network succesfully and are thrilled to see it working with a 95% reliability. All of these achievements would never have been possible if our team was not communicating and always uplifting one another. We are grateful to have each other as commrades for this project.