

Rapport de Soutenance Finale - Projet S3 EPITA
4D Generates

4D Generates

Gustave HERVÉ, Léo SAMBROOK, Arthur HAMARD et Briac GUELLEC

Décembre 2022

4D



Table des matières

1	Introduction	3
1.1	Notre Groupe	3
2	Répartition des Charges	3
3	Fonctionnement de l'OCR Sudoku	3
3.1	Prétraitement de l'image	3
3.1.1	Passage en niveaux de gris	3
3.1.2	Flou gaussien	4
3.1.3	Binarisation adaptive	6
3.1.4	Opérations morphologiques	7
3.1.5	Détection des blobs	9
3.2	Détection de la grille et des cases	11
3.2.1	Transformée de Hough	11
3.2.2	Transformée homographique	13
3.2.3	Extraction des cases	18
3.2.4	Normalisation des cases	18
3.3	Réseau de Neurones	20
3.3.1	XOr	20
3.3.2	Reconnaissance de Chiffres	22
3.4	Solveur de Sudoku	27
3.5	Affichage du résultat	29
3.6	Interface Graphique	30
4	Ce qui a changé depuis la première soutenance	33
4.1	Prétraitement et détection de lignes	33
4.1.1	Algorithme de Canny	34
4.1.2	Détection de la grille et des cases	34
5	Conclusion	35

1 Introduction

1.1 Notre Groupe

Notre groupe est constitué de : Gustave HERVÉ, Léo SAMBROOK, Arthur HAMARD et Briac GUELLEC. Nous sommes heureux, dans ce rapport, de vous présenter la version finale de notre projet de S3 consistant en un OCR.

2 Répartition des Charges

Tâches	Responsable
Neural Network + Solver	Léo
Traitement image + Détection de grille	Gustave
Rotation + Transformée Homographique	Briac
Interface + Sauvegarde résultat	Arthur

3 Fonctionnement de l'OCR Sudoku

3.1 Prétraitement de l'image

L'objectif de cette partie est d'appliquer différents traitements sur l'image pour en simplifier la détection des éléments qui nous intéressent, à savoir la grille dans un premier temps, puis les chiffres par la suite.

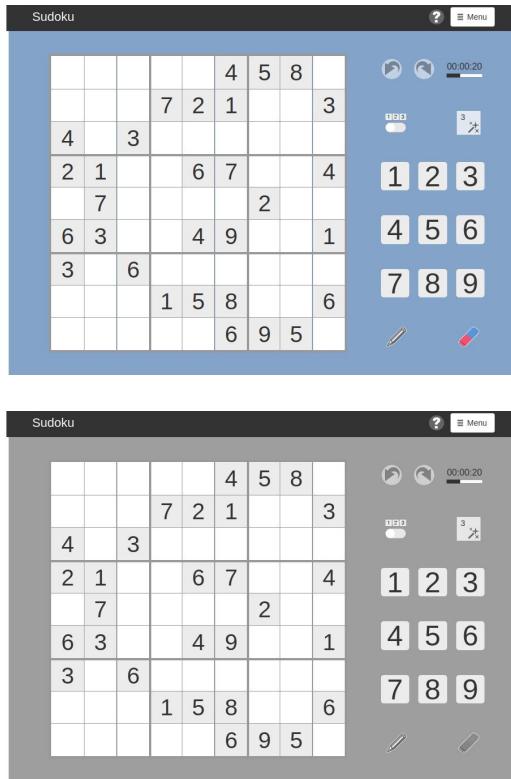
3.1.1 Passage en niveaux de gris

Les informations de couleur n'étant pas utiles pour détecter la grille, la première étape consiste à retirer les couleurs pour ainsi convertir les pixels vers leur homologue en niveaux de gris. Pour ce faire, nous pouvons simplement récupérer les valeurs RGB de chaque pixel puis calculer ce que nous appellerons l'intensité du pixel selon la formule suivante :

$$\text{Intensity} = 0.3 * R + 0.59 * G + 0.11 * B$$

Cette valeur d'intensité remplace alors les trois valeurs R, G et B du pixel en question, chaque pixel de l'image devient alors tel que pour tout pixel de coordonnées i,j :

$$R_{ij} = B_{ij} = G_{ij} = \text{Intensity}_{ij}$$



Passage en noir et blanc de l'image

3.1.2 Flou gaussien

Avant de commencer l'étape de binarisation, nous floutons l'image pour réduire au maximum le bruit lié à une luminosité hétérogène. Pour ce faire nous calculons en premier lieu une matrice de Gauss de taille (n,p) telle que pour chaque élément composant la matrice :

$$G(x, y) = e^{-\frac{x^2+y^2}{2\sigma^2}} \text{ avec } \sigma = 0.3 * ((size - 1) * 0.5 - 1) + 0.8$$

Les valeurs de cette matrice sont ensuite normalisées pour avoir une somme de chaque coefficient qui vaut 1. Cela permet d'éviter un assombrissement ou une augmentation de la luminosité de l'image pendant le processus. Cette matrice nous sert de filtre de convolution que l'on applique sur l'image pour obtenir le flou recherché.

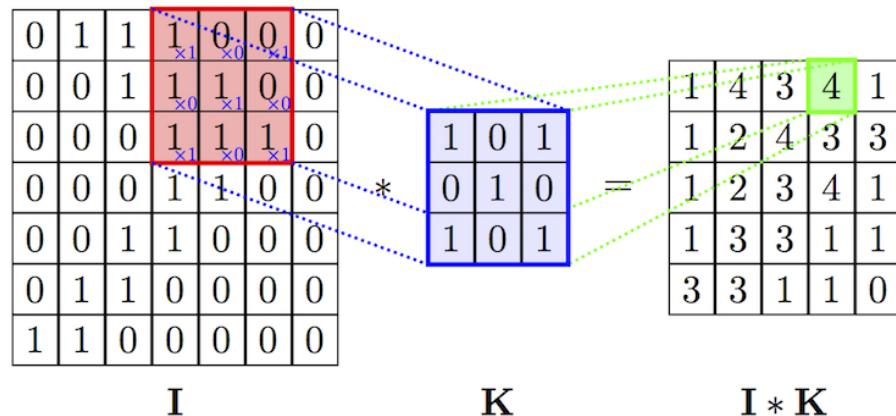


FIGURE 1 – Illustration de la convolution de matrice

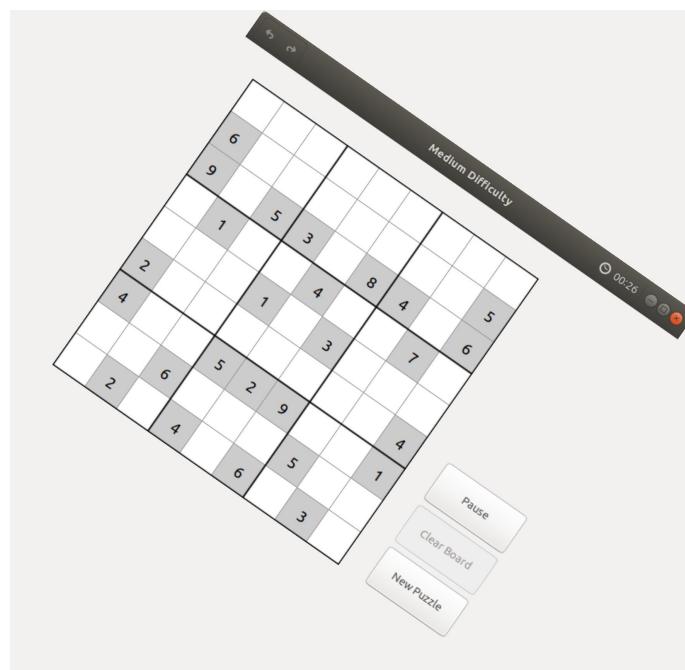


FIGURE 2 – Image originale

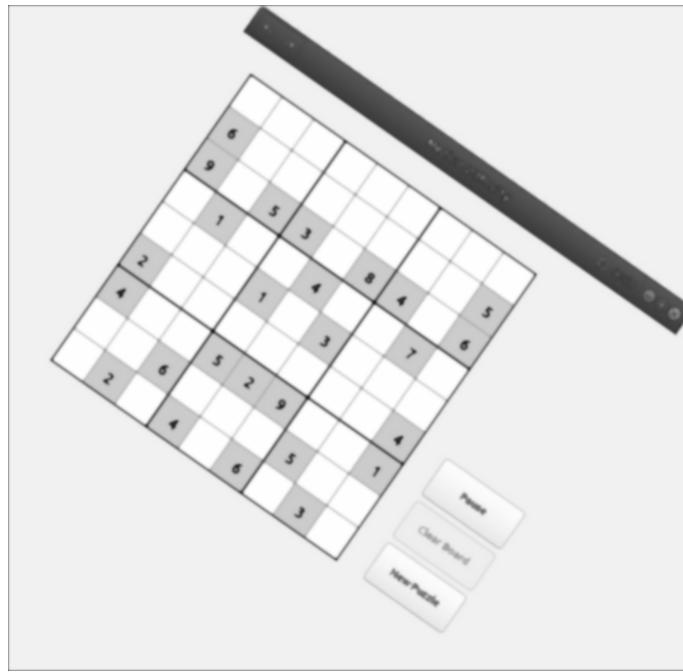


FIGURE 3 – Image après en passage noir et blanc et flou gaussien

3.1.3 Binarisation adaptive

Pour étudier l'image, il est nécessaire de la simplifier au point de ne plus avoir que des pixels blancs ou des pixels noirs. Cette étape est la binarisation de l'image et pour se faire nous utilisons une fois de plus une matrice de Gauss qui nous permet par convolution de calculer cette fois-ci un seuil d'intensité pour chaque pixel. L'avantage de cette méthode est qu'elle est locale et non globale comme peut être la méthode d'Otsu, ce qui permet une bien plus grande robustesse au bruit. Ce seuil est calculé en fonction du voisinage de 13 pixels alentours à celui que nous souhaitons traiter. Nous calculons la somme des valeurs d'intensité des pixels alentours multipliées par la matrice de Gauss, puis nous définissons le seuil tel que pour un pixel (i,j) :

$$Seuil_{ij} = Somme_{ij} - c$$

avec c une valeur d'offset fixée à 3. Ce-dernier nous permet d'éviter les situations où le fond de l'image deviendrait entièrement blanc et viendrait masquer la grille, rendant impossible toute tentative de détection. Une fois la valeur de seuil du pixel étudié calculée, il suffit de regarder si le pixel dispose d'une valeur d'intensité supérieure à ce seuil. Si oui, le pixel est un contour valide et son pixel est mis en blanc, dans le cas contraire, son pixel est mis en noir.

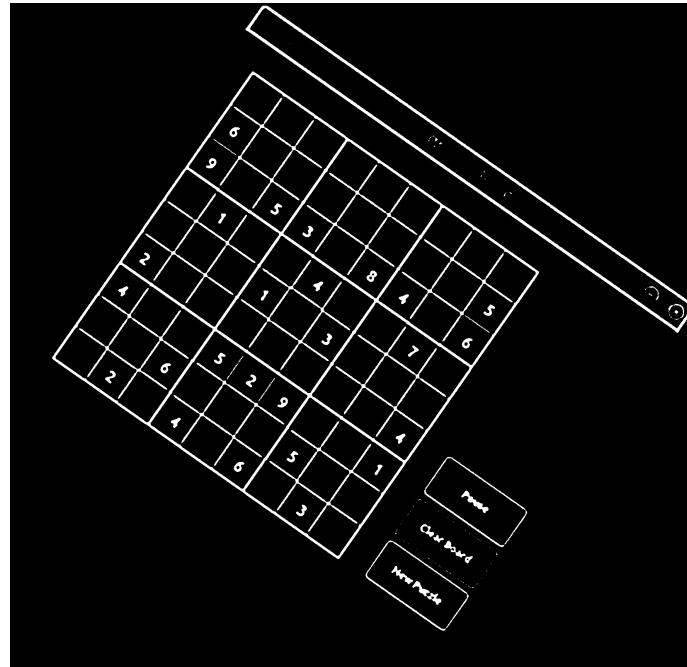


FIGURE 4 – Résultat de la binarisation

3.1.4 Opérations morphologiques

Pour faciliter les opérations qui suivent, nous disposons de quelques opérations morphologiques qui ont pour but d'accentuer ou au contraire réduire les pixels blancs sur une image binarisée. Ces opération fonctionnent par convolution sur une zone de voisinage de pixel de taille prédéfinie. Ces opérations sont les suivantes :



FIGURE 5 – Image de référence utilisée pour illustrer les opérations morphologiques

Dilatation : L'objectif de la dilatation est d'accentuer les bords blancs d'une image binarisée. Pour chaque pixel, nous cherchons dans le voisinage de celui-ci le pixel ayant la plus grande valeur d'intensité. Cette valeur maximum remplace alors celle du pixel en question. Cette opération est notamment utile pour l'étape suivante de détection de la grille par blob.



FIGURE 6 – Image après dilatation, on remarque des bordures plus épaisses

Erosion : L'érosion est l'opération inverse, au lieu de chercher le pixel de plus grande intensité aux alentours, nous cherchons celui ayant la plus petite intensité. Cette opération nous permet de rétablir l'épaisseur originale de l'image après dilatation pour réduire la marge d'erreur lors de la détection des coins de la grille.



FIGURE 7 – Image après érosion, on remarque des bordures plus fines

Combinées, ces deux opérations peuvent former les deux suivantes en fonction de l'ordre d'exécution :

Fermeture : La fermeture consiste en une dilatation suivie d'une érosion. Elle permet notamment de fermer des "trous" jonchant une image, ce qui nous est une fois de plus utile pour réduire la marge d'erreur de la détection de la grille.



FIGURE 8 – La fermeture permet de combler les lignes discontinues

Ouverture : L'ouverture est l'opération inverse de la fermeture, il s'agit d'une érosion suivie d'une dilatation. L'ouverture permet d'éliminer des bruits parasites jonchant une image, ce qui combiné avec le flou gaussien nous permet de plus aisément nettoyer l'image.



FIGURE 9 – Le bruit disparait après l'ouverture

3.1.5 Détection des blobs

Une fois l'image traitée, il est temps de détecter où se situe la grille. Pour ce faire, nous appliquons un parcours de l'image proche d'un parcours de graphe. L'objectif est de déterminer l'amas de pixels blancs adjacents le plus grand de l'image (ce que nous appellerons "blob"). L'algorithme que nous utilisons pour faire cela est le remplissage par diffusion ou flood-filling en anglais.

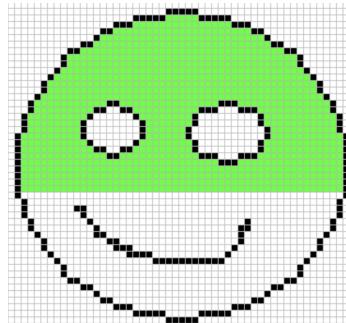


FIGURE 10 – Le remplissage par diffusion est notamment utile pour délimiter des zones de couleur unie

Pour appliquer cet algorithme, nous utilisons une structure de pile dans laquelle on va empiler le premier pixel blanc que nous trouvons dans l'image. A partir de ce premier pixel, notre objectif va être de trouver tous les pixels adjacents dans quatre directions : nord, est, sud, ouest. A chaque fois que l'on dépile un pixel, on le marque d'une valeur différente pour éviter qu'il soit visité plusieurs fois. Une fois le parcours du blob terminé (c'est-à-dire lorsqu'il n'y a plus de voisins non visités) nous renvoyons deux informations à son sujet : la position du pixel de départ du parcours ainsi que le nombre de pixels parcourus. Nous parcourons tout l'image selon ce principe jusqu'à ce qu'il ne reste plus aucun pixel blanc non marqué, puis nous utilisons la position du pixel du blob avec le plus de pixels pour retrouver la grille et la transférer vers une image vide.

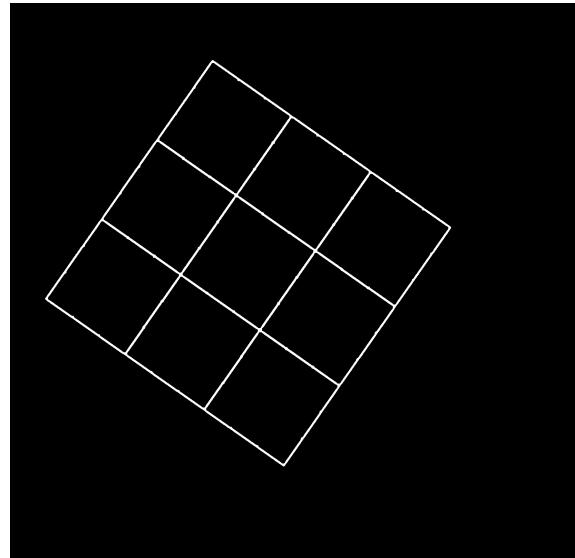


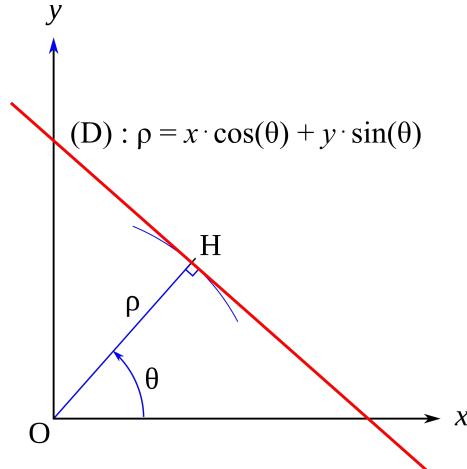
FIGURE 11 – Résultat du flood-filling

3.2 Détection de la grille et des cases

A ce stade, nous avons notre grille visuellement mais nous devons encore trouvé les coordonnées (x,y) des quatre coins de la grille pour pouvoir appliquer une rotation automatique sur l'image ainsi qu'en extraire le contenu des cases. Pour ce faire, nous chercons d'abord à détecter les droites composant le blob.

3.2.1 Transformée de Hough

Pour détecter les lignes présentes dans l'image du blob, nous utilisons la transformée de Hough. Cet algorithme utilise les coordonnées polaires pour identifier les lignes en fonction de deux paramètres : leur distance par rapport à l'origine de l'image (le paramètre ρ) et l'angle qu'elles forment avec l'axe X (le paramètre θ).



(a) La transformée de Hough utilise une représentation polaire

Pour déterminer les lignes d'une image, une matrice H de taille $\rho * \theta$ est formée et initialisée à 0 pour chacune de ses valeurs. Pour chacun des pixels formant un contour de l'image (pixel blanc), on calcule pour θ allant de 0 à 360 le paramètre ρ tel que pour un pixel de coordonnées (x,y) :

$$\rho = x * \cos \theta + y * \sin \theta$$

on incrémenté alors la valeur de $H[\rho][\theta]$.

Une fois la matrice remplie, nous allons chercher les indices ρ et θ des maximums locaux, c'est-à-dire les points avec les valeurs les plus hautes. Pour ce faire, nous calculons une valeur seuil T :

$$T = 0.5 * Max$$

avec Max la plus grande valeur contenue dans la matrice H .

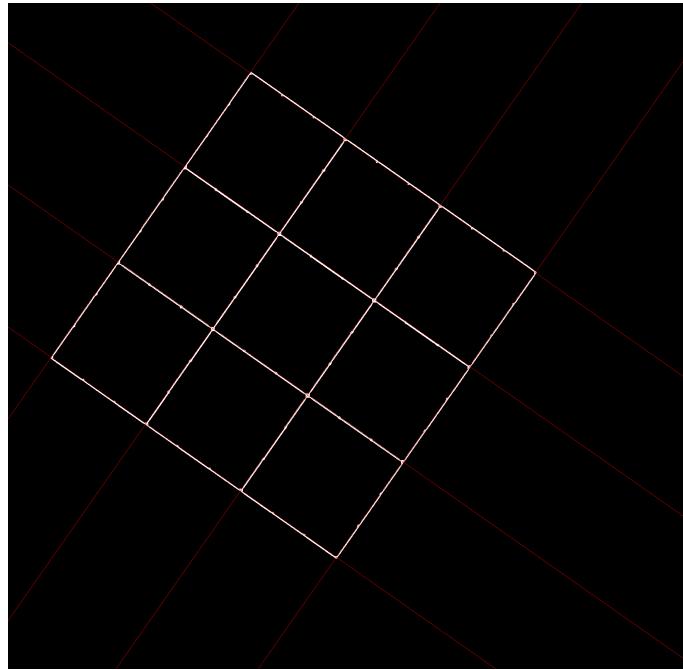


FIGURE 13 – Lignes détectées par la transformées de Hough

Il s'agit maintenant de trouver le plus grand carré formé par les points d'intersections de ces lignes. Pour ce faire, nous itérons sur chaque droite trouvée par Hough et cherchons une autre ligne qui entre en intersection avec celle-ci. Nous prenons cette autre ligne et cherchons également une autre droite distincte entrant en intersection. Nous répétons le processus trois fois (les deux derniers points d'intersection peuvent se trouver avec la troisième ligne choisie) et vérifions s'il s'agit bien d'un carré. Si oui, nous enregistrons la longueur de son côté et continuons la recherche. Nous gardons à la fin le carré de côté le plus grand.

Une fois les coordonnées de la grille déterminés, nous pouvons appliquer une correction de perspective pour isoler la grille.

3.2.2 Transformée homographique

La transformée homographique est un algorithme ayant de nombreux aspects pratiques et parmi ceux-ci on peut citer la correction de perspective et la rotation automatique. Cette fonction prend en paramètres les 4 coins de la grille et selon ceux-ci renverra une image contenant une grille "droite". La transformée homographique n'est, en réalité, qu'une suite de systèmes à résoudre sous forme de matrices. La matrice sur laquelle se base cet algorithme et qui permet de trouver pour chaque

pixel au coordonnées (x,y) son équivalent dans la nouvelle image au coordonnées (x',y') est la suivante :

Soit $(a1, a2, a3, b1, b2, b3, c1, c2, c3) \in \mathbf{R}^9$,

$$\begin{aligned} \begin{bmatrix} h.x' \\ h.y' \\ h \end{bmatrix} &= \begin{bmatrix} a1 & a2 & a3 \\ b1 & b2 & b3 \\ c1 & c2 & c3 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \\ \implies \begin{bmatrix} h.x' \\ h.y' \\ h \end{bmatrix} &= \begin{bmatrix} a1.x + a2.y + a3 \\ b1.x + b2.y + b3 \\ c1.x + c2.y + c3 \end{bmatrix} \\ \implies \begin{bmatrix} x' \\ y' \\ \end{bmatrix} &= \begin{bmatrix} \frac{a1.x + a2.y + a3}{c1.x + c2.y + c3} \\ \frac{b1.x + b2.y + b3}{c1.x + c2.y + c3} \\ \end{bmatrix} \end{aligned}$$

Les 9 constantes posées ci-dessus ne correspondent à aucun pixel de l'image mais sont essentielles pour trouver les coordonnées de chaque pixel dans l'image résultante. Il convient donc, avant d'utiliser cette formule, de trouver la valeur de ces 9 constantes. Pour ce faire, nous avons besoin de connaître les 4 coins de la grille et de calculer la taille de la nouvelle image.

Cette dernière s'obtient en calculant, à partir des 4 coins de la grille, le côté le plus long.

Pour faciliter la compréhension des calculs qui suivront, posons deux matrices : "old" et "new" qui contiennent respectivement les coordonnées des coins de la grille et les coordonnées de la nouvelle image.

$$\text{old} = \begin{bmatrix} x1 \\ y1 \\ x2 \\ y2 \\ x3 \\ y3 \\ x4 \\ y4 \end{bmatrix} \quad \text{new} = \begin{bmatrix} 0 \\ 0 \\ size \\ 0 \\ size \\ size \\ 0 \\ size \end{bmatrix}$$

Comme évoqué précédemment, pour pouvoir calculer les coordonnées de chaque pixel dans la nouvelle image, il nous faut déterminer la valeur des constantes.

Posons premièrement P, une matrice carrée de dimension 9 telle que :

$$P = \begin{bmatrix} -old[0] & -old[1] & -1 & 0 & 0 & 0 & old[0] * new[0] & old[1] * new[0] & new[0] \\ 0 & 0 & 0 & -old[0] & -old[1] & -1 & old[0] * new[1] & old[1] * new[1] & new[1] \\ -old[2] & -old[3] & -1 & 0 & 0 & 0 & old[2] * new[2] & old[3] * new[2] & new[2] \\ 0 & 0 & 0 & -old[2] & -old[3] & -1 & old[2] * new[3] & old[3] * new[3] & new[3] \\ -old[4] & -old[5] & -1 & 0 & 0 & 0 & old[4] * new[4] & old[5] * new[4] & new[4] \\ 0 & 0 & 0 & -old[4] & -old[5] & -1 & old[4] * new[5] & old[5] * new[5] & new[5] \\ -old[6] & -old[7] & -1 & 0 & 0 & 0 & old[6] * new[6] & old[7] * new[6] & new[6] \\ 0 & 0 & 0 & -old[6] & -old[7] & -1 & old[6] * new[7] & old[7] * new[7] & new[7] \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Posons une autre matrice R :

$$R = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Il nous faut maintenant une fonction capable d'inverser une matrice. Cette fonction nécessite elle-même plusieurs fonctions : une première qui calcule le déterminant, une seconde qui retourne la matrice adjointe et une dernière qui retourne la matrice de cofacteurs.

Une fois encore posons une matrice H telle que :

$$H = P^{-1}.R$$

Nous avons une H matrice colonne, transformons la en une matrice carrée de dimension 3 que nous noterons M. Inversons cette matrice et nous obtenons une matrice de dimension 3 contenant les constantes recherchées.

$$M^{-1} = \begin{bmatrix} a1 & a2 & a3 \\ b1 & b2 & b3 \\ c1 & c2 & c3 \end{bmatrix}$$

Après avoir trouvé cette matrice il ne nous reste plus qu'à parcourir tous les pixels de l'image et trouver grâce à la première équation leur équivalent dans la nouvelle image.

Appliquer cet algorithme à une image corrige la perspective et la rotationne automatiquement comme le montrent les images ci-dessous.

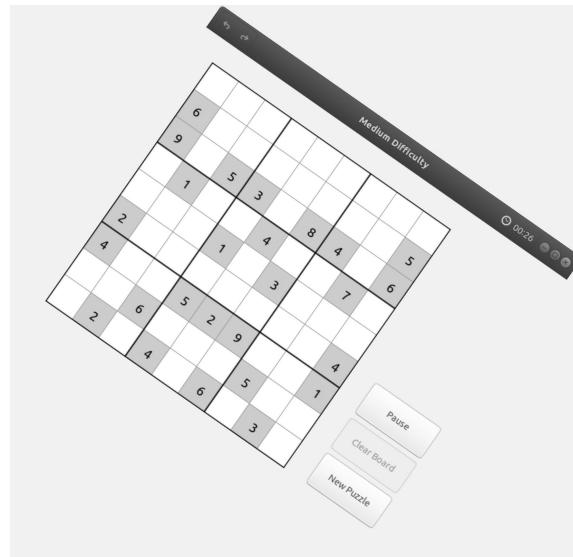


image before Homographic Transform

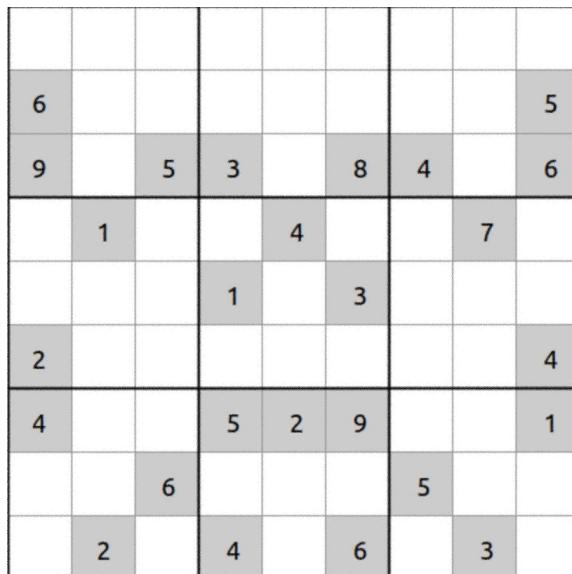


image after Homographic Transform

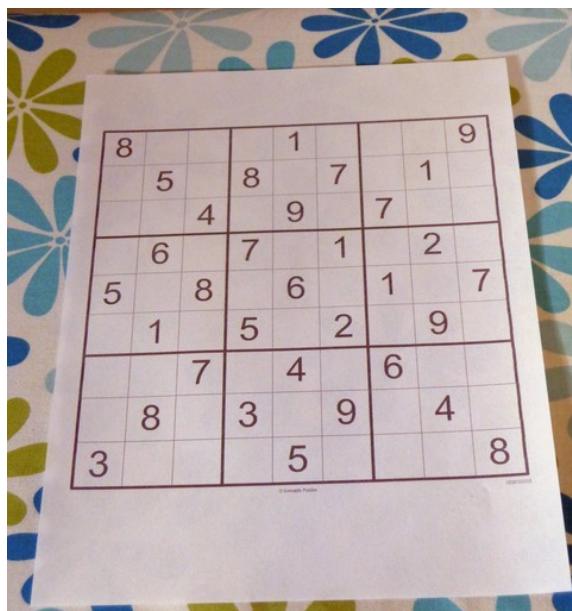


image before Homographic Transform

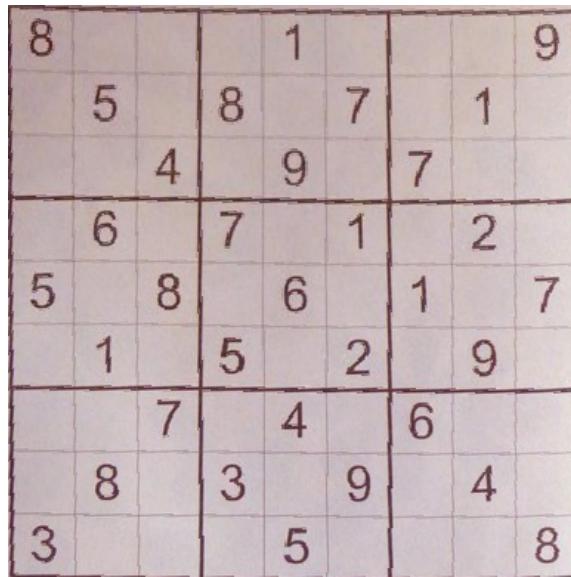


image after Homographic Transform

3.2.3 Extraction des cases

Pour pouvoir envoyer le contenu des 81 cases au réseau de neurones, nous devons extraire les cases de façon individuelles et les enregistrer dans un dossier temporaire. Nous utilisons le résultat de la correction de perspective pour déterminer la longueur du carré formant une case en pixels :

$$l = L/9 \text{ avec } L \text{ la longueur du carré résultant de la transformée homographique}$$

Il s'agit ensuite simplement de copier chaque case vers une image et de l'enregistrer.

3.2.4 Normalisation des cases

Néanmoins, avant de pouvoir envoyer ces images découpées au réseau de neurones il est nécessaire de les normaliser. Les images doivent en effet suivre un format particulier : celles-ci doivent contenir le chiffre en noir sur fond blanc, centré et prenant toute la hauteur de l'image. Les images doivent de plus être dans une résolution 28x28 pixels.

Pour pouvoir centrer correctement le chiffre dans une case de 28x28 pixels, nous devons le détecter. Nous utilisons le même algorithme que celui utilisé pour la grille, l'algorithme de détection par remplissage. Une fois le chiffre détecté, nous calculons la "bounding box" du chiffre, c'est-à-dire le rectangle minimum permettant d'entourer le chiffre.

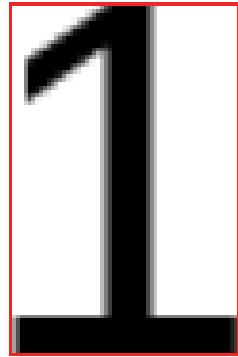


FIGURE 14 – Exemple de bounding box en rouge délimitant un 1

Pour obtenir la bounding box il suffit de garder en mémoire les pixels visités ayant les plus grandes valeurs x et y. On peut faire de même pour ceux ayant les plus petites valeurs. Nous avons ensuite quatre points formant la bounding box :

$$(x_{min}, y_{max}), (x_{max}, y_{max}), (x_{max}, y_{min}), (x_{min}, y_{min})$$

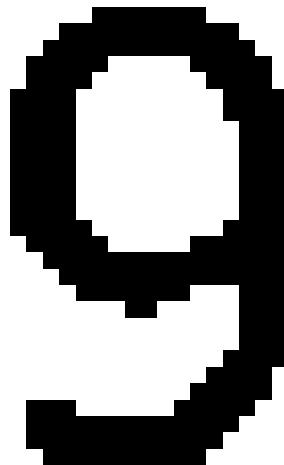


FIGURE 15 – Exemple d'image normalisée

3.3 Réseau de Neurones

3.3.1 XOr

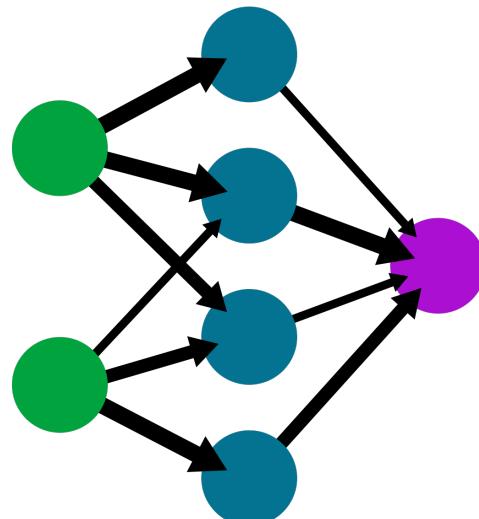
L'architecture :

Notre réseau de neurones XOR est composé de 3 couches. La première, la couche d'entrée ou noeuds d'entrée (Input nodes en anglais) est composée de 2 noeuds car dans l'opérateur logique OU EXCLUSIF, il y a deux entrées ([00],[01],[10],[11]). La couche intermédiaire (2ème couche, hidden nodes en anglais) quant à elle est composée de 4 noeuds (Plus de précision et rapidité que 2). Et pour finir la dernière couche (output nodes en anglais) n'est composée que d'un seul noeud car il n'y a qu'un seul résultat (soit 1 soit 0). Nous appliquons ensuite la propagation vers l'avant en partant des deux premiers neurones appartenant à la couche d'entrée. Et nous appliquons la rétropropagation en partant de la dernière couche.

Notre réseau de neurones peut être représenté de la manière suivante :

A simple neural network

input layer	hidden layer	output layer
----------------	-----------------	-----------------



(a) Schéma de notre réseau de neurones

Comme il est indiqué sur le schéma, chaque noeud est relié par des "câbles" appelés "arêtes" et sur ces "arêtes" sont attribués un poids qui va multiplier les valeurs transmises d'un noeud à un autre. "L'intelligence" d'un réseau de neurone est basé sur l'apprentissage et ce sont ces "poids" qui vont permettre à un réseau de neurones de donner des bons résultats. Il est donc important de les attribuer et de les mettre à jour correctement. C'est pour cela que nous utilisons deux méthodes : la propagation vers l'avant et la rétropropagation.

Propagation vers l'avant :

La propagation vers l'avant consiste à partir des neurones de la couche d'entrée pour aller jusqu'à/aux couche(s) de sortie tout en passant par chaque neurone de chaque couche dans le but de mettre à jour leur poids.

Lorsqu'un neurone reçoit une valeur, il va pouvoir la modifier avec une fonction mathématique avant de l'envoyer vers son successeur. Cette fonction mathématique s'appelle la fonction "d'activation" et son but est de déterminer si oui ou non, il doit transmettre la valeur reçue au prochain. Si la valeur passe au prochain alors elle est proche de 1, sinon elle est proche de 0.

Chaque neurone a la possibilité d'ajouter à l'entrée de la fonction d'activation un "biais" qui est une petite valeur multipliée par le poids. Ces biais permettent aux neurones d'avoir une influence sur la fonction d'activation. Il existe plusieurs fonctions d'activations, celle que nous utilisons s'appelle la sigmoïde :

$$S(z) = 1/(1 + \exp(-z))$$

L'avantage de cette fonction est qu'elle renvoie des valeurs dans l'intervalle [0 ; 1] ce qui permet d'interpréter la sortie du neurone comme une probabilité. De plus, comme sa dérivée est assez simple, elle évite de ralentir encore plus notre fonction.

Rétropropagation :

Le but de la "rétropropagation", ou la propagation en partant de la sortie, est de minimiser les erreurs de poids. Pour calculer les erreurs nous utilisons une fonction mathématique très simple qui consiste à soustraire le résultat obtenu par le résultat attendu et multiplier le résultat par la fonction dérivée de la fonction sigmoïde appelée DSigmoïde :

$$S'(z) = S(z) * (1 - S(z))$$

Une fois que la valeur "d'erreur" a été calculée, on multiplie chaque poixs par leur valeur d'erreur.

Sauvegarde de la mémoire :

Nous avons dit plus tôt que l'efficacité et la précision d'un réseau de neurones dépendait des poids et c'est pour cela qu'à la fin de notre réseau de neurones nous sauvegardons les poids de chaque arête dans un fichier. Lorsque nous relançons le réseau de neurones, on ré-attribue les valeurs sauvegardées à chaque poids car sinon le réseau va se réinitialiser et perdre tout son "entraînement". Lorsque le réseau de neurones est lancé pour la première fois, on attribue des valeurs aléatoires aux poids.

3.3.2 Reconnaissance de Chiffres

L'architecture :

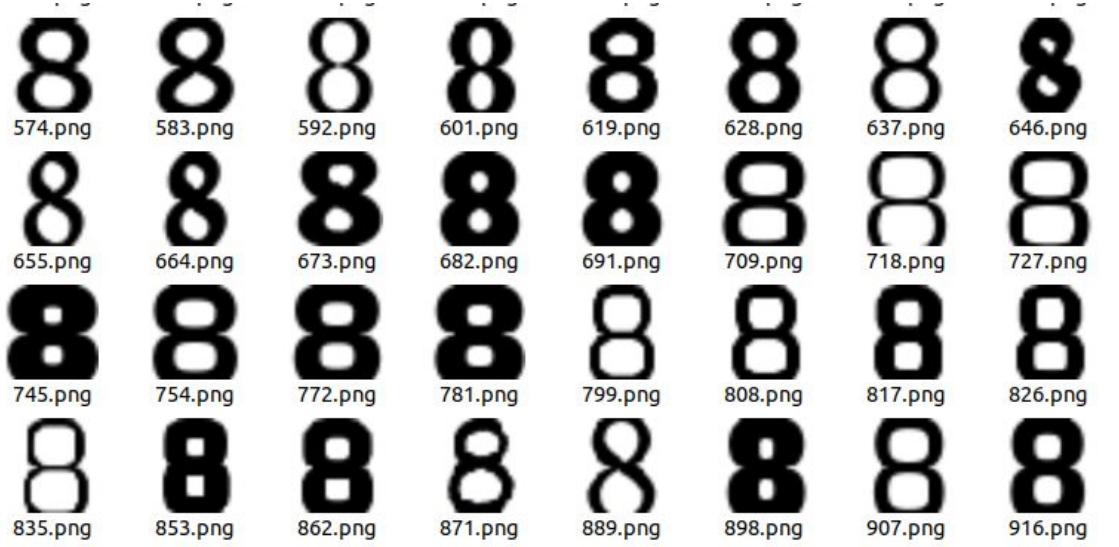
Nous avons commencé à implémenter le réseau de neurones qui va détecter le chiffre sur une image. Celui-ci est capable de s'entraîner sans problème avec un jeu d'images allant de 0 à 9. La manière dont il s'entraîne est quasi identique à celle du XOR mais comporte tout de même quelques différences. En effet, au début du programme nous convertissons les pixels d'une image de chiffres en 0 et 1 (0 pour pixel noir et 1 pixel blanc) chaque pixel va correspondre à un neurone d'entrée et, comme les images sont de taille 28x28, il y a 784 neurones dans la première couche. De plus, la couche intermédiaire passe de 3 neurones à 175 pour plus de précision. Pour finir, la dernière couche, c'est à dire celle de sortie, possède 10 neurones car il y a 10 chiffres possibles dans un sudoku (0,1,2,3,4,5,6,7,8,9) et le neurone qui possède la plus grande valeur en sortie correspond au chiffre de l'image. Par exemple si c'est le neurone de sortie numéro 3 qui possède la plus grande valeur alors le réseau de neurones en déduit que le chiffre est un 3.

Entrainement :

Contrairement au XOR, il existe une quasi infinité d'images de chiffres différents et de ce fait nous ne pouvons définir de manière exacte les valeurs qui vont être dans

les neurones d'entrées. Pour que le réseau de neurones puisse lire et déduire chaque chiffre de chaque image du sudoku avec la plus petite marge d'erreur possible il nous a fallu trouver un "data-set" avec beaucoup d'images diverses et variées. Au début nous pensions entraîner le réseau de neurones avec les images sorties de l'extraction de cases, sauf que comme nous n'avons que 7 images de sudoku, le data-set sera trop petit pour entraîner efficacement. Nous avons donc décidé de nous orienter vers un script python qui va récupérer 250 images de chaque chiffre écrit avec des polices d'écriture différentes et en taille 28x28.

Voici un exemple :



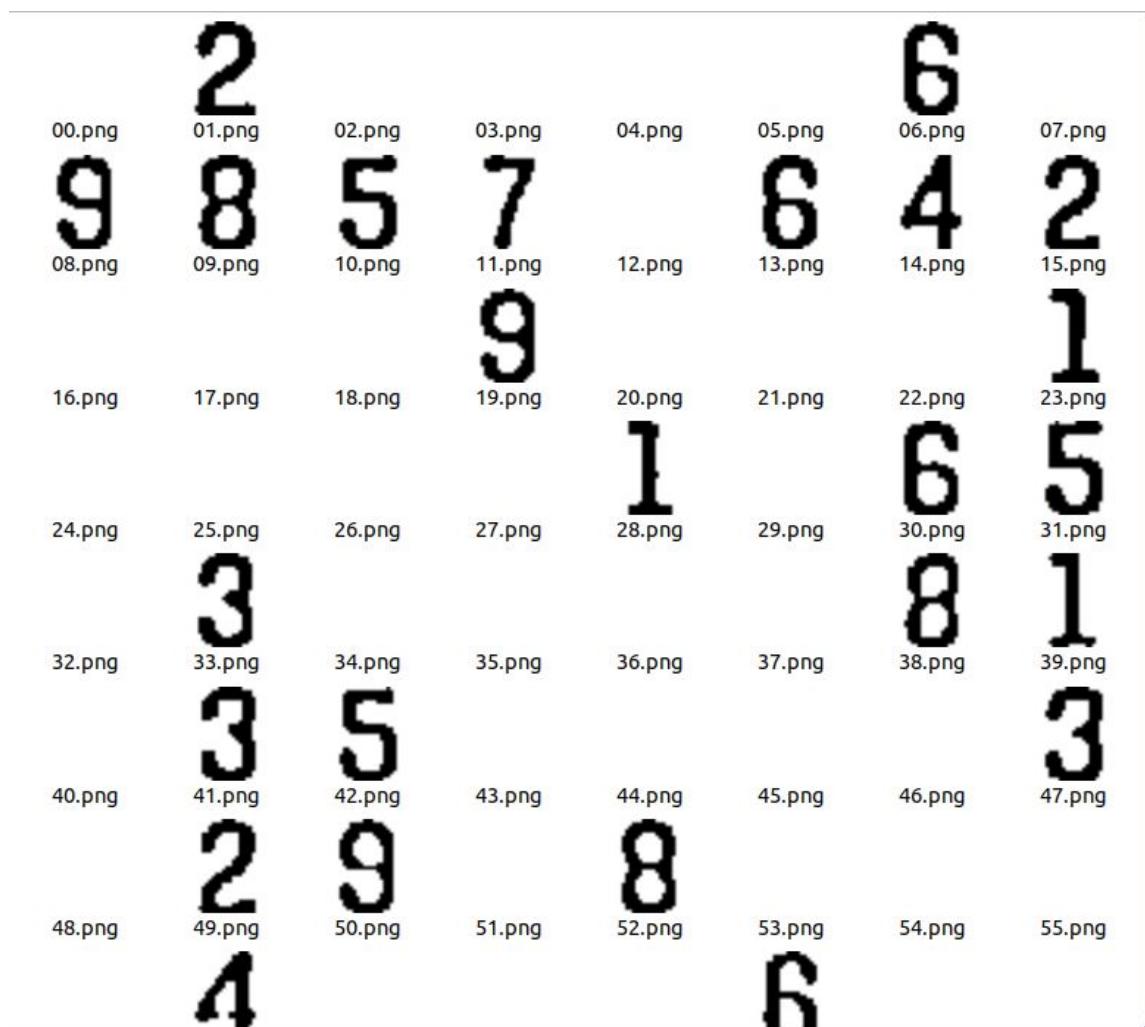
Quelques chiffres 8 extraits par le script python

Une fois ce nouveau "data set" acquis, il fallait que le réseau de neurones puisse toutes les lire. Pour ce faire nous avons parcouru toutes les images de chaque dossier (dossier d'images 1, 2, ..., 9), nous les avons mis dans des array et ainsi appelé 250 fois le réseau de neurones avec de nouvelles images de chaque chiffre à chaque fois.

Lecture du dossier :

Une fois que l'implémentation de l'entraînement était terminée il nous fallait implémenter la fonction qui allait récupérer chaque image présente dans le dossier généré par l'extraction des cases. L'implémentation de cette fonction est très similaire à celle

de l'entraînement. Nous utilisons la librairie <dirent> qui permet d'ouvrir un dossier et de parcourir chaque image. Comme la librairie <dirent> parcourt les images de manière aléatoire, nous lisons les chiffres présents dans l'image (00.png, 01.png etc..) et nous ajoutons chacune de ces images dans un array pour ensuite les donner au réseau de neurones.



Exemple de dossier

```
final/Image_01_SET3/38.png
final/Image_01_SET3/32.png
final/Image_01_SET3/17.png
final/Image_01_SET3/70.png
final/Image_01_SET3/19.png
final/Image_01_SET3/52.png
final/Image_01_SET3/03.png
final/Image_01_SET3/64.png
final/Image_01_SET3/79.png
final/Image_01_SET3/20.png
final/Image_01_SET3/18.png
final/Image_01_SET3/06.png
final/Image_01_SET3/00.png
final/Image_01_SET3/58.png
final/Image_01_SET3/04.png
final/Image_01_SET3/71.png
```

Exemple de parcours de dossier

Détection des chiffres :

Passons maintenant à la partie où le réseau de neurones doit déduire le chiffre présent sur l'image. Dans un premier temps, l'IA va récupérer tous les poids et biais présents dans le fichier "Brain" composé de 130k lignes. Ensuite, en fonction de l'image il va rentrer 0 ou 1 dans les neurones d'entrées et suite à cela il déduit le résultat avec le "front propagation" seulement et retourne directement le résultat.

```
training_data/8/943.png
Image is a 8
training_data/8/151.png
Image is a 8
training_data/8/2491.png
Image is a 8
training_data/8/2518.png
Image is a 8
```

Exemple de résultat

Bonus - chiffres manuscrits :

En guise de "bonus" nous avons fait en sorte que le réseau de neurones puisse aussi lire et déduire des chiffres manuscrits. La difficulté avec les images écrites à la main c'est que les pixels ne sont jamais blanc ou noir ce qui est problématique pour les valeurs d'entrées du réseau de neurones. De ce fait, nous avons implémenté une fonction qui va permettre d'arrondir un nombre entre 0 et 255 en 1 ou 0.



Images manuscrites

Transformation en fichier texte :

Comme expliqué précédemment, chaque résultat que le réseau de neurones renvoie est stocké dans un array, suite à quoi nous parcourons cet array et nous écrivons chaque chiffre dans un fichier appelé "grid_00" (image ci-dessous) que nous allons donner par la suite au solver pour le résoudre.

3.4 Solveur de Sudoku

Le solveur de sudoku est la pierre angulaire du projet. Il s'agit du programme qui, à partir d'une matrice de chiffres, va être capable de résoudre un sudoku. Il prendra donc en paramètre un fichier comportant les chiffres du sudoku précédemment scannés à l'aide du réseau de neurones. Ici, les zéros sont représentés avec des points.

```
$ cat grid_00
... .4 58.
... 721 ..3
4.3 ... ...
...
21. .67 ..4
.7. ... 2..
63. .49 ..1
...
3.6 ... ...
... 158 ..6
... .6 95.
```

Entrée du programme Solver.

```
$ cat grid_00.result
127 634 589
589 721 643
463 985 127

218 567 394
974 813 265
635 249 871

356 492 718
792 158 436
841 376 952
```

Sortie du Solver

Fonctionnement du Programme

Ce solver, afin de compléter la grille, va utiliser un algorithme de backtracking, ou de retour sur trace. L'algorithme consiste à sélectionner une case à remplir, et d'essayer récursivement de résoudre le sudoku avec le numéro sélectionné pour cette case, en appliquant l'algorithme sur les autres cases, prenant en compte les changements précédents. Si il est impossible de le résoudre dans ces conditions, alors le système va revenir légèrement en arrière sur ses décisions afin de sortir du blocage (d'où le nom de retour sur trace). Il va, par la suite, réessayer sur la case initiale avec de nouveaux paramètres, et sur les autres cases, de manière récursive.

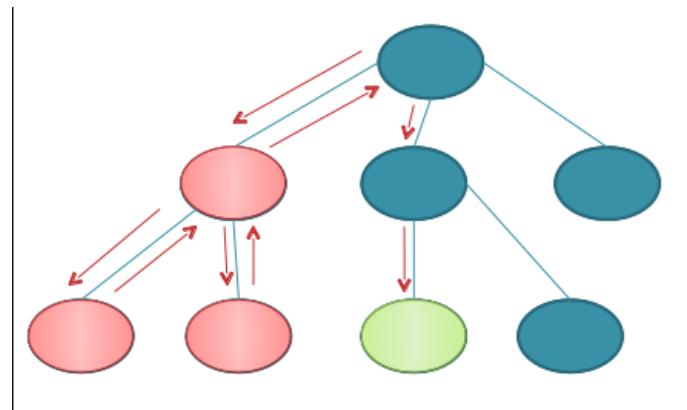


Illustration de l'algorithme de backtracking, les noeuds représentant les différentes

possibilités de résolution du sudoku.

3.5 Affichage du résultat

Pour afficher le résultat nous avons utilisé un template de grille vide ainsi que des templates de cases ayant une taille d'exactement 150 par 150 pixels contenant tous les chiffres en rouge et en noir. Le programme prend en paramètres deux fichiers textes qui représentent la grille non résolue et la grille après résolution. Le principe de l'algorithme est de parcourir ces deux fichiers simultanément et de regarder si le chiffre était présent dans la grille non résolue ou non. Si le chiffre n'était pas présent dans la grille non résolue alors il faut copier dans la grille résultat le chiffre rouge se trouvant à cette place dans la grille résolue. Bien entendu, si le chiffre était déjà présent dans la grille non résolue, il suffit de le copier à la bonne place dans la grille résultat. Ceci permet de mettre en évidence pour l'utilisateur les chiffres qui n'étaient pas dans la grille de base.

1	2	7	6	3	4	5	8	9
5	8	9	7	2	1	6	4	3
4	6	3	9	8	5	1	2	7
2	1	8	5	6	7	3	9	4
9	7	4	8	1	3	2	6	5
6	3	5	2	4	9	8	7	1
3	5	6	4	9	2	7	1	8
7	9	2	1	5	8	4	3	6
8	4	1	3	7	6	9	5	2

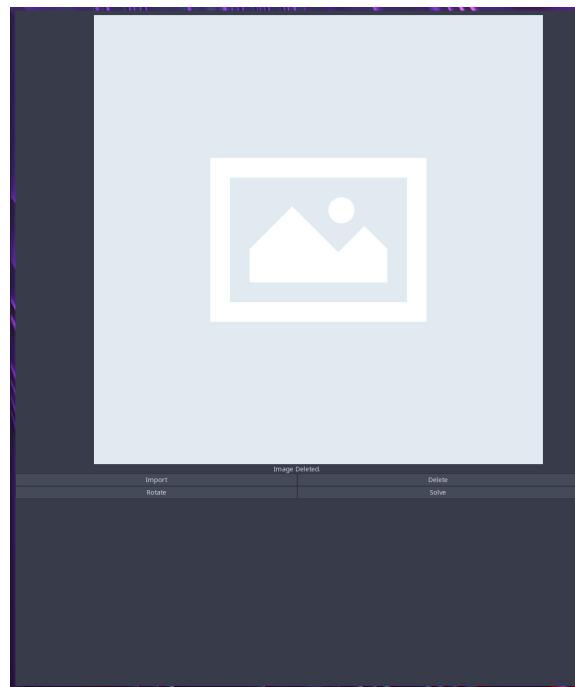
Grille résolue avec mise en évidence des changements

3.6 Interface Graphique

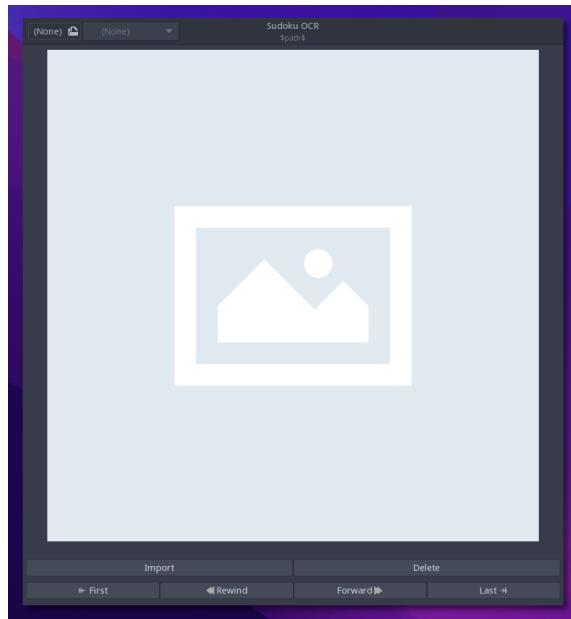
Afin de proposer une agréable expérience à l'utilisateur, nous avons mis en place une interface graphique permettant de manipuler toutes les fonctions de l'application. Cette dernière va permettre de gérer manuellement les aspects techniques vus précédemment tels que la rotation ou la résolution de la grille de sudoku.

Construction du GUI

Pour ce dernier, nous avons utilisé la librairie GTK, qui permet de créer des applications de manière simple en C. De plus, nous avons incorporé une structure créée à l'aide de l'outil Glade, permettant de réaliser le cadrage, les boutons et les labels (presque) sans lignes de codes. GTK restant néanmoins nécessaire à l'exécution des scripts au contact des boutons, nous n'utilisons Glade que pour la forme de l'application et non le fond.



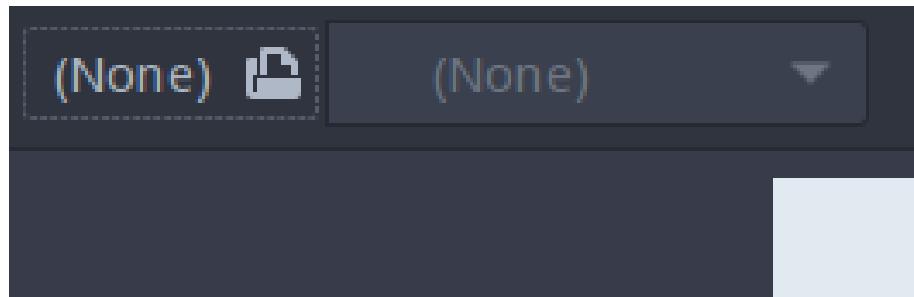
Version Initiale du GUI



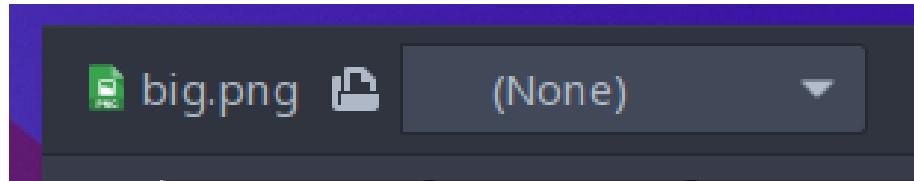
Version Finale du GUI

Import et Export

Premièrement, afin d'importer l'image du sudoku que l'on souhaite résoudre, un bouton d'importation se situe dans le coin du GUI. Ce bouton ouvre une vue des dossiers où les fichiers affichés sont restreints aux formats images. A côté de ce bouton se situe une fonction pour sauvegarder le sudoku résolu. Ce dernier utilise le même principe que le bouton d'import, et va afficher une vue des dossiers de la machine. Pour sauvegarder l'image, il suffit de sélectionner le dossier et de valider. Le bouton de sauvegarde n'est utilisable qu'après la résolution du sudoku et sauvegarde l'image finale.



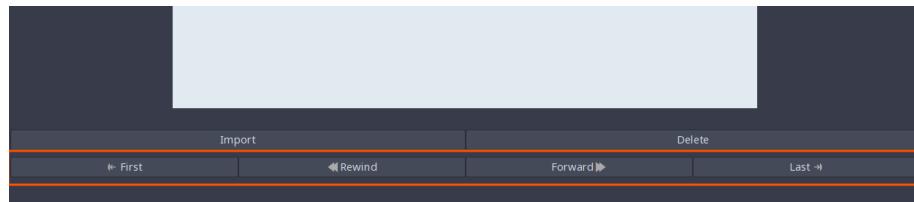
Boutons d'Importation et de Sauvegarde, ce dernier étant désactivé car la dernière étape n'est pas initialisée



Boutons d'Importation et de Sauvegarde, ce dernier étant actif car la dernière étape est initialisée

Gestion des Etapes

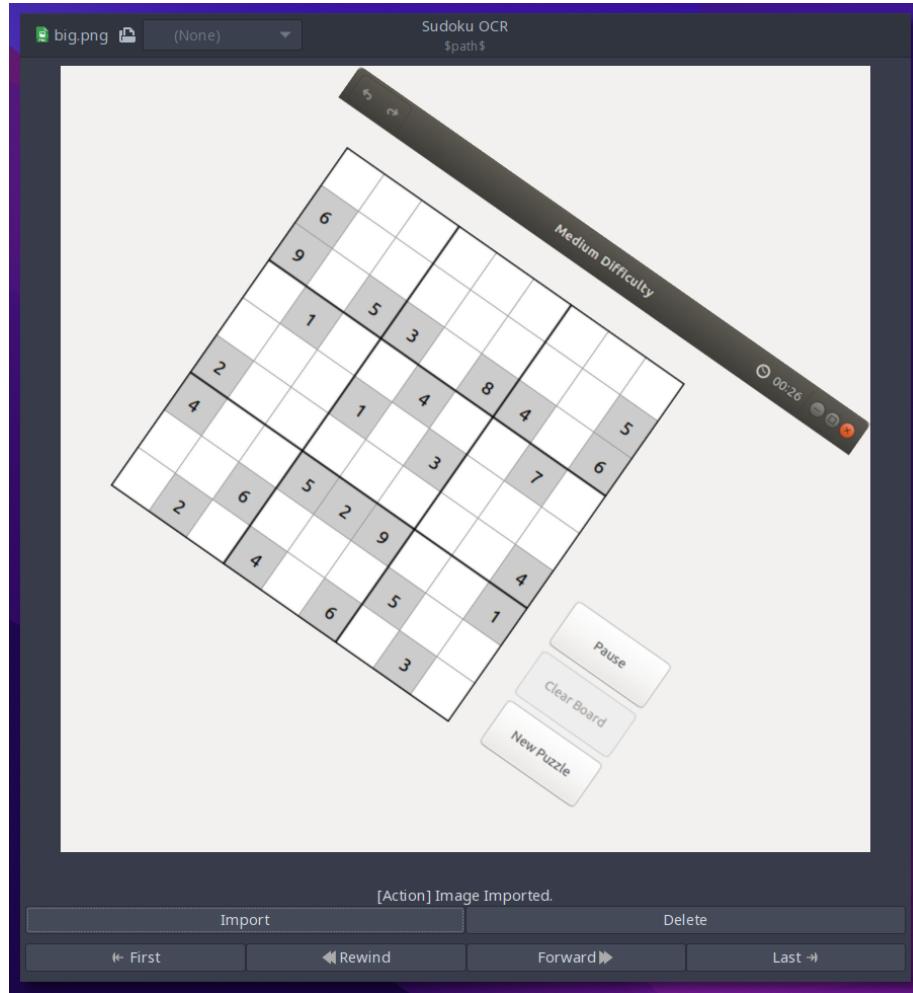
Afin de résoudre la grille, nous avons procédé étape par étape aux exécutions des fonctions au sein de l'application. Commencant par la mise en niveau de gris de l'image, et se finissant par la génération de l'image finale, ces 12 étapes sont gérées par un accumulateur qui est modifié par l'appel de 4 boutons. Un bouton pour avancer dans la resolution, un pour reculer, un pour accéder à l'image finale si cette dernière est générée, et un pour revenir à l'étape 1. L'accumulateur permet par ailleurs de savoir le niveau d'avancement actuel, et la prochaine action à effectuer.



Boutons de sélection des étapes

Communication avec l'Utilisateur

Afin de transmettre toutes erreurs à l'utilisateur, nous avons implémenté un label permettant d'afficher les avertissements ou problèmes. Ce dernier communique également le changement d'étape ainsi que des messages informatifs tel que l'importation de fichiers ou leur exportation.



Exemple d'importation d'image avec message du Label

4 Ce qui a changé depuis la première soutenance

4.1 Prétraitement et détection de lignes

Nous avons changé de façon significative le traitement de l'image ainsi que la détection de grille depuis la première soutenance.

4.1.1 Algorithme de Canny

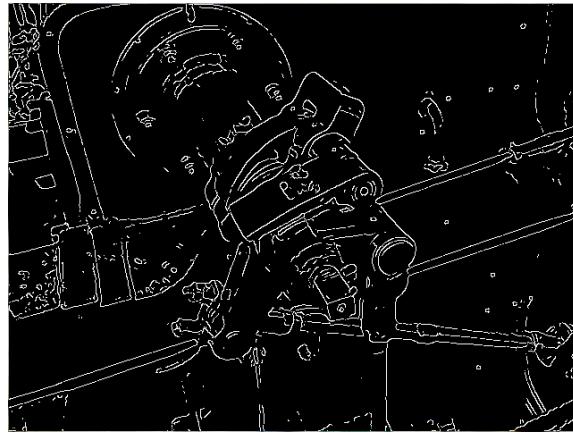


FIGURE 17 – L'algorithme de Canny permet d'isoler des bords très fins

Pour l'étape de binarisation de l'image, nous avons décidé de retirer l'algorithme de Canny au profit d'un seuillage par matrice de Gauss. La première raison à cela étant qu'il s'agit d'un algorithme comportant de nombreuses étapes lourdes et difficiles à maintenir. Les bords fins que Canny renvoie ne sont par ailleurs pas adaptés à notre application et rendent plus difficile la détection de la grille. Le filtre de Canny étant très sensible au taux de flou applique à l'image, il était alors très difficile de trouver des paramètres pour le flou fonctionnant sur suffisamment d'images à la fois. Nous avions de plus des problèmes de bruit très important sur certaines photos dont la luminosité n'était pas homogène.

4.1.2 Détection de la grille et des cases

Auparavant nous utilisions Hough transform dans l'optique de détecter non seulement la grille mais également les lignes séparant les cases. Cette approche était compliquée à mettre en place car il fallait être en mesure de trouver la vraie grille parmi toutes les lignes détectées par Hough. Une première stratégie que nous avions implémentée était d'essayer de trouver 10 lignes parallèles et de distance égales horizontales, de même pour les lignes verticales. Cela pouvait fonctionner sur les images numériques simples mais il était impossible d'appliquer cet algorithme sur de vraies

photos de sudoku prises avec une caméra de par la présence de "fausses lignes" causée par les textes et cadres alentours. Une autre approche que nous avons tentée était de chercher à détecter le plus grand carré formé par les lignes sans passer par une détection de blob comme c'est le cas maintenant. Nous rencontrons le même problème avec cette fois des "faux carrés" détectés et pris comme grille.

5 Conclusion

Nous espérons que notre projet vous aura convaincu. Nous sommes heureux de la tournure qu'a pris ce travail de groupe, et bien que tout ne soit pas parfaitement optimisé ni forcément opérationnel sur toutes les images, nous sommes