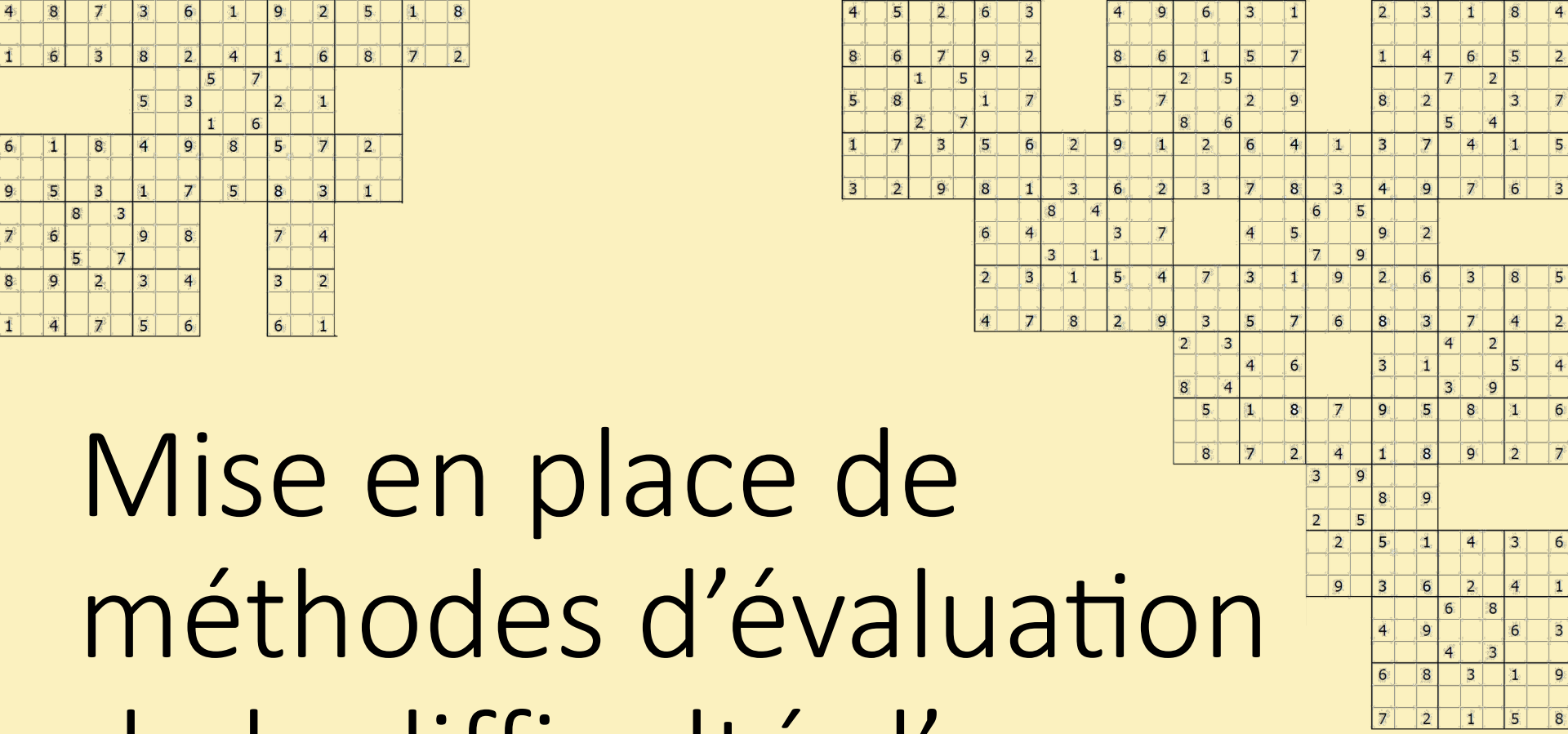


# Mise en place de méthodes d'évaluation de la difficulté d'une grille de sudoku.

TIPE 2025

Félix Landreau- MPI

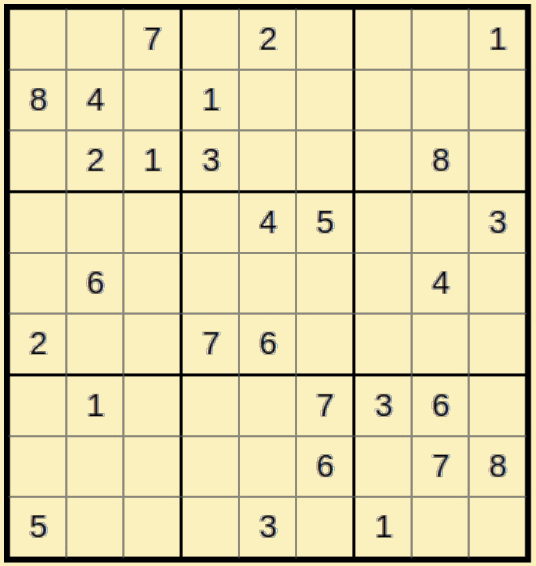
Numéro de candidat : 13185

# Motivation

Pas d'échelle de difficulté normalisée

Évaluation empirique par les éditeurs

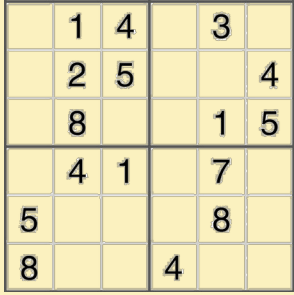
- Critères visuels
- Algorithmes



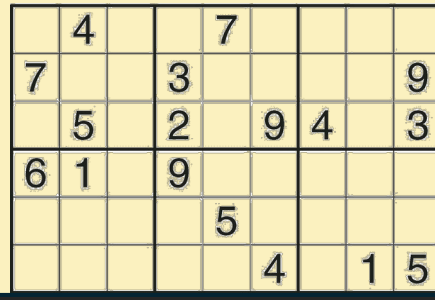
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 7 | | 2 | | | | 1 |
| 8 | 4 | | 1 | | | | | |
| | 2 | 1 | 3 | | | | 8 | |
| | | | | 4 | 5 | | | 3 |
| | 6 | | | | | | 4 | |
| 2 | | | 7 | 6 | | | | |
| | 1 | | | | 7 | 3 | 6 | |
| | | | | | 6 | | 7 | 8 |
| 5 | | | | 3 | | 1 | | |

Grille de sudoku

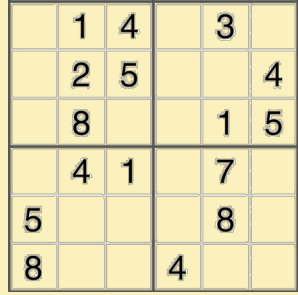
# Problématique




Pour mesurer la difficulté humaine d'une grille de sudoku
quels critères visuels peut-on retenir pour une évaluation faite par un humain,
et quels algorithmes peut-on élaborer pour une évaluation faite par un ordinateur ?
Comment étendre ces résultats à d'autres jeux de logique ?



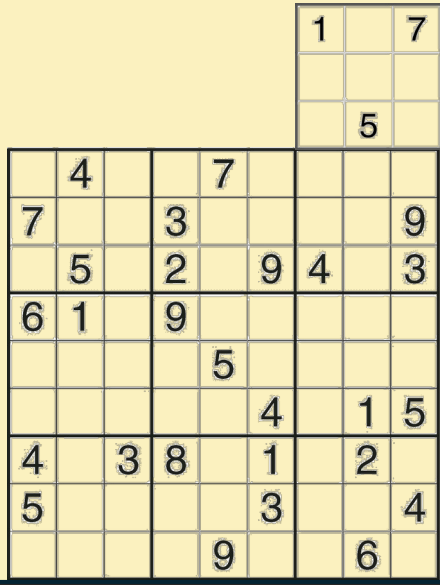
# Plan




1- Critères visuels
   - A – Constat initial
   - B – Critères étudiés
   - C – Résultats

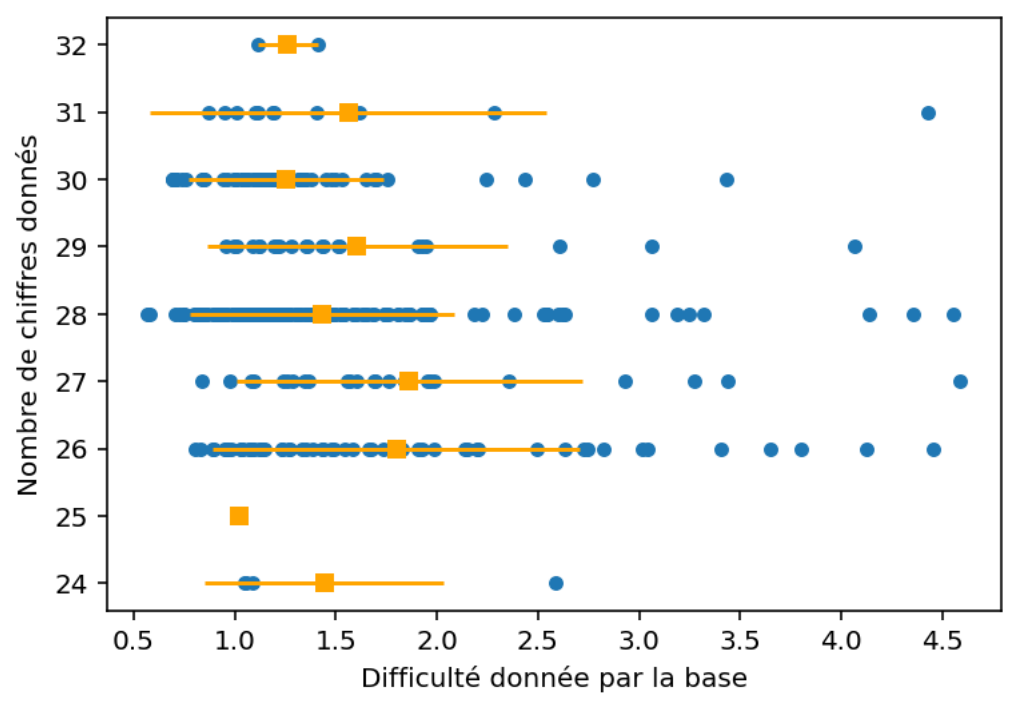
2- Évaluation par résolution par techniques humaines
   - A – Calcul des techniques
   - B – Étalonnage de l'évaluateur
   - C – Résultats

3- Lien avec le problème SAT
   - A – Conversion
   - B – Résolution
   - C – Évaluation



# 1.A – Critères visuels – Constat initial

Méthode instinctive : nombre de chiffres donnés




Base de données 1 : https://github.com/synnwang/sudoku_dataset_difficulty

# 1.B – Critères visuels étudiés

- Nombre total de candidats
- Répartition géographique des chiffres donnés
- Répartition entre valeurs des chiffres donnés

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 7 | | 2 | | | | 1 |
| 8 | 4 | | 1 | | | | | |
| | 2 | 1 | 3 | | | | 8 | |
| | | | | 4 | 5 | | | 3 |
| | 6 | | | | | | 4 | |
| 2 | | | 7 | 6 | | | | |
| | 1 | | | | 7 | 3 | 6 | |
| | | | | | 6 | | 7 | 8 |
| 5 | | | | 3 | | 1 | | |

Facile

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

Difficile

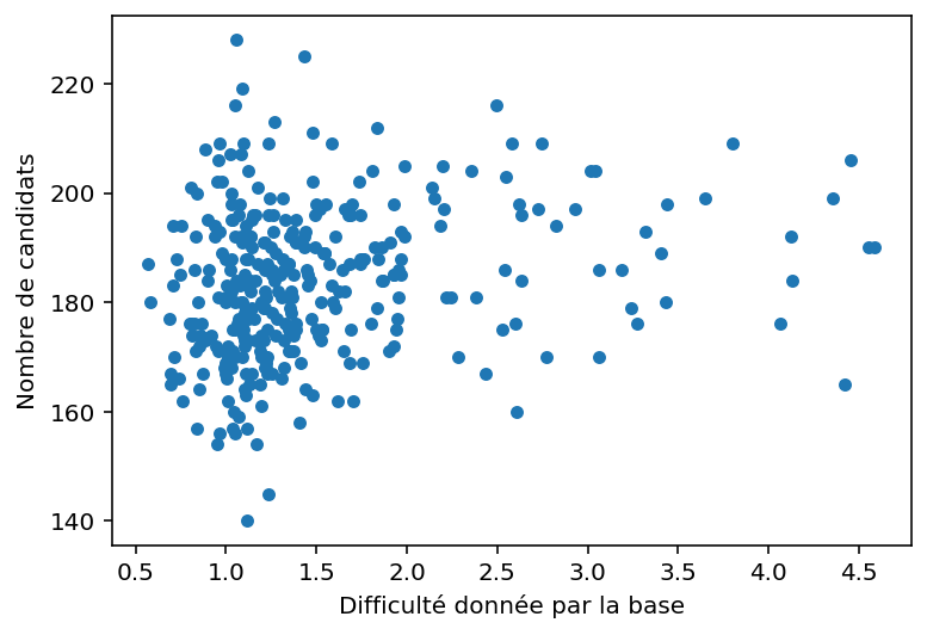
# 1.C – Critères visuels - Résultats

Nombre total de candidats

| | | | |
|---|---|---|---|
| 2 3 | 1 | 1 2 | 4 |
| 2 3 | 4 | 1 2 | 1 3 |
| 4 | 2 | 3 | 1 |
| 1 | 3 | 4 | 2 |

Score = 14

Base de données 1




Corrélation : 0,21

# 2.A – Calcul des techniques

Critères de difficulté d'un sudoku :

- ~~Critères visuels~~
- Techniques de résolution

  (détaillées en annexe)

$$\overbrace{S}^{grille} \rightarrow \overbrace{(n_1, n_2, \dots n_p)}^{occurences\ des\ techniques} \rightarrow Difficulté(S) = \sum_{j=1}^{p} \underbrace{\alpha_j}_{poids}\ n_j$$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 7 | | 2 | | | | 1 |
| 8 | 4 | | 1 | | | | | |
| | 2 | 1 | 3 | | | | 8 | |
| | | | | 4 | 5 | | | 3 |
| | 6 | | | | | | 4 | |
| 2 | | | 7 | 6 | | | | |
| | 1 | | | | 7 | 3 | 6 | |
| | | | | | 6 | | 7 | 8 |
| 5 | 7 | | | 3 | | 1 | | |

Singleton caché

# 2.A – Calcul des techniques

Conception d'un solveur progressif en langage C :

- Techniques humaines
- Techniques de difficulté minimale

Hypothèse :
Les $p$ techniques sont fournies par ordre croissant de difficulté.

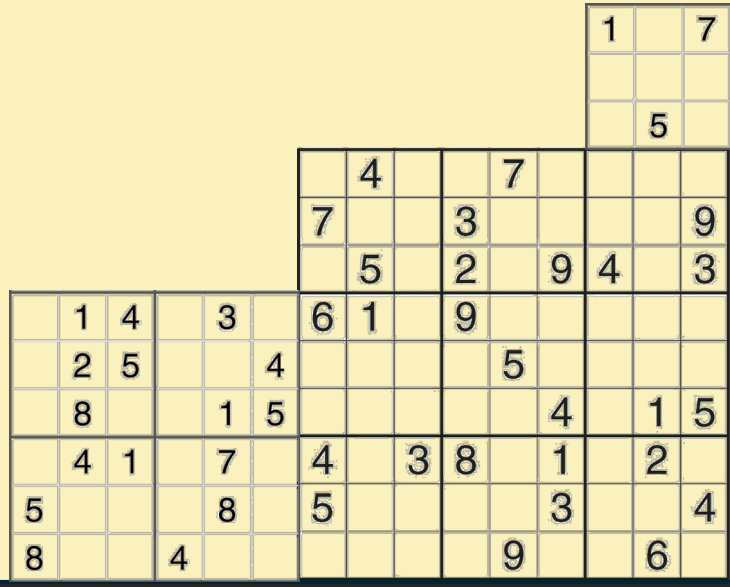
# 2.B – Étalonnage de l'évaluateur

$$\overbrace{S_i}^{grille} \rightarrow \overbrace{(n_{i,1}, n_{i,2}, \dots n_{i,p})}^{occurences\ des\ techniques} \rightarrow Difficulté(S_i) = \sum_{j=1}^{p} \underbrace{\alpha_j}_{poids} n_{i,j}$$

$p = 13$ techniques, $m \simeq 300$ grilles

Ajustement des poids des différentes techniques

Hypothèse :
La difficulté est linéaire en le nombre d'utilisations des techniques.



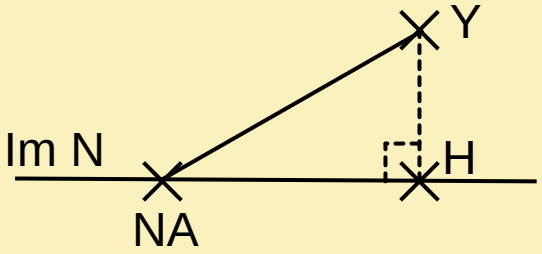
# 2.B – Étalonnage de l'évaluateur

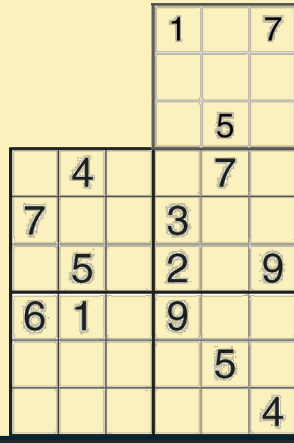
## Méthode analytique

$$\overbrace{S_i}^{grille} \rightarrow \overbrace{(n_{i,1}, n_{i,2}, \dots n_{i,p})}^{occurences\ des\ techniques} \rightarrow Difficulté(S_i) = \sum_{j=1}^{p} \underbrace{\alpha_j}_{poids}\, n_{i,j}$$

$$N = (n_{i,j})_{1 \le i \le m, 1 \le j \le p}, \quad A = (\alpha_j)_{1 \le j \le p}, \quad Y = (y_i)_{1 \le i \le m}$$

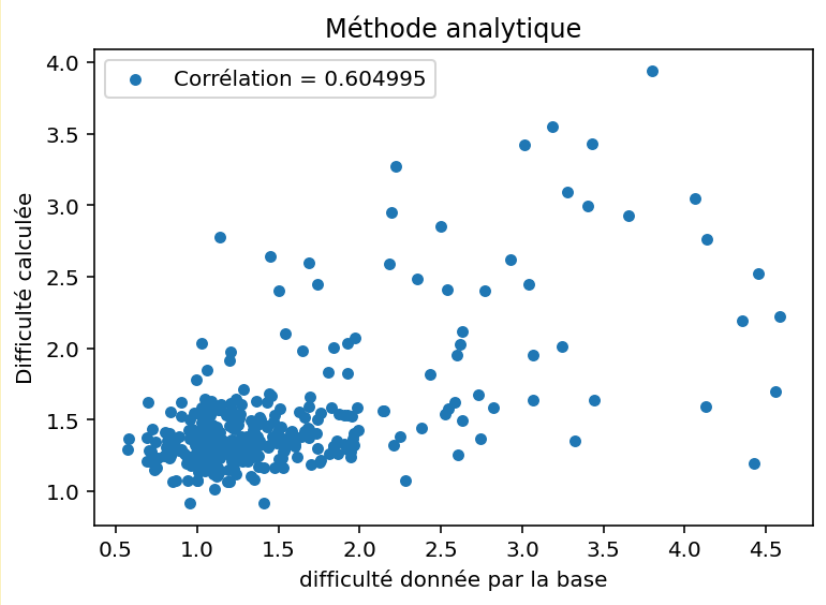
On souhaite minimiser $\|NA - Y\|_2$



# 2.C- Résultats

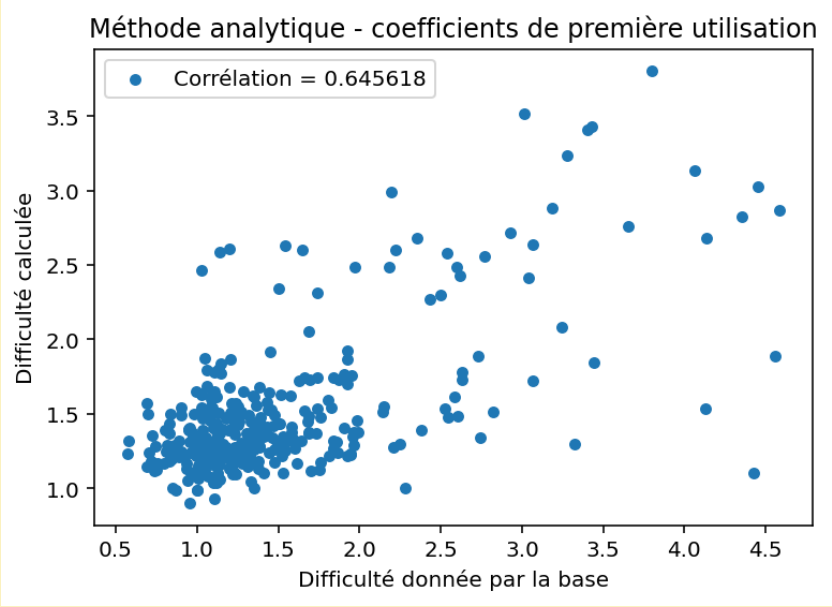
## Base de données n°1




| Technique | Coefficient |
|---|---|
| Last Free Cell | -0,01 |
| Hidden Single | 0,06 |
| Naked Single | 0,07 |
| Naked Pair | 0,08 |
| Naked Triple | 0,2 |
| Pointing Pair | -0,04 |
| Hidden Pair | 0,87 |
| Hidden Triple | 2,5 |
| Box-Line reduction | 1,7 |
| X-Wing | 0,9 |
| Y-Wing | -0,1 |
| Swordfish | 6 |
| Backtracking | 0,7 |

# 2.C- Résultats

## Base de données n°1

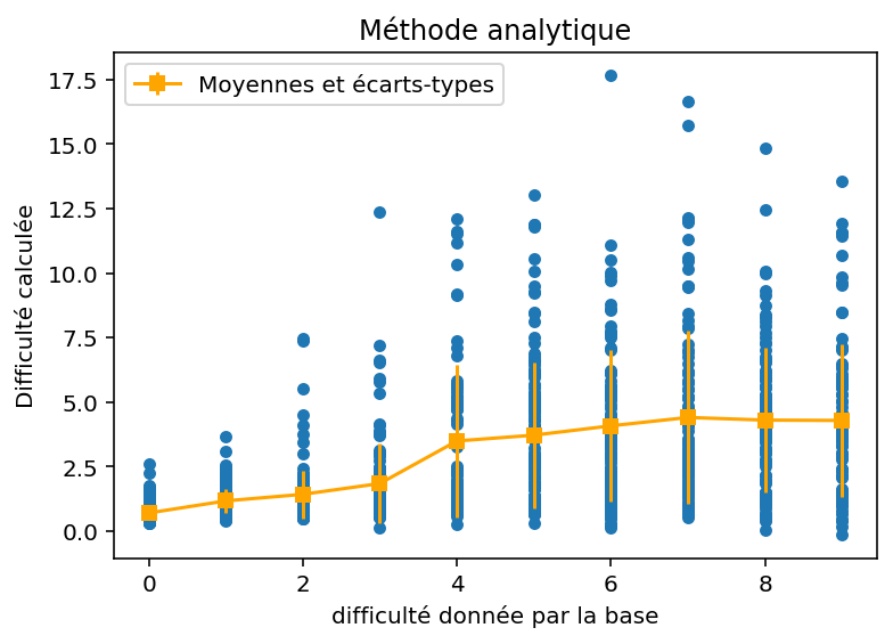



$$Difficulté(S) = \sum_{i=1}^{p} \alpha_i n_i + \sum_{i=1}^{p} \beta_i 1_{n_i>0}$$

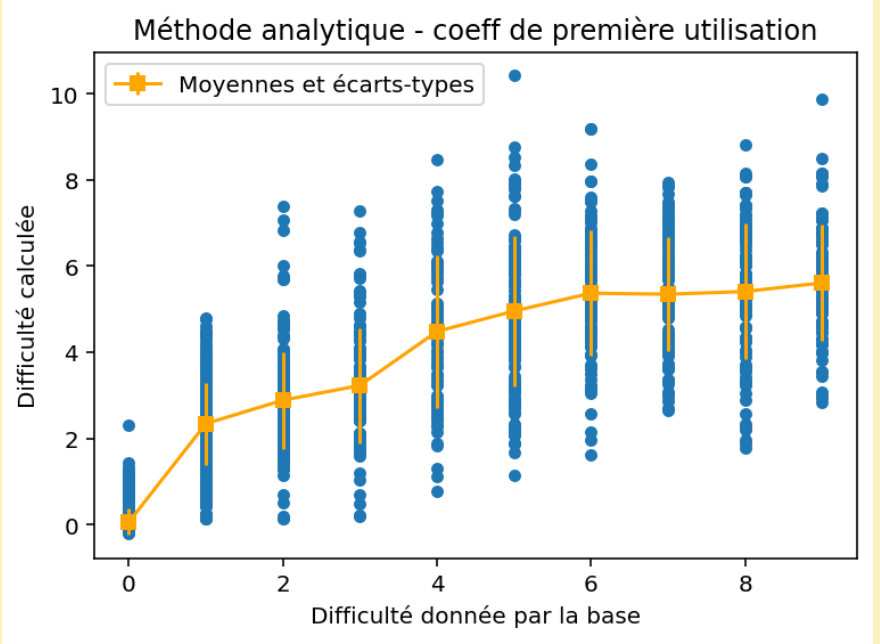
| Technique | Coefficient | Coefficient de première utilisation |
|---|---|---|
| Last Free Cell | 0,05 | -0,7 |
| Hidden Single | 0,08 | -0,7 |
| Naked Single | -0,04 | 0,5 |
| Naked Pair | 0,05 | -0,06 |
| Naked Triple | 0,1 | -0,03 |
| Pointing Pair | -0,04 | 0,7 |
| Hidden Pair | 0,02 | 1,5 |
| Hidden Triple | 8 | 7 |
| Box-Line Reduction | 0,4 | 0,4 |
| X-Wing | -2,7 | -2,7 |
| Y-Wing | -9 | 13 |
| Swordfish | 0,1 | 0,1 |
| Backtracking | -3 | 10 |

# 2.C- Résultats

Base de données n°2, difficultés de 0 à 9




Corrélation : 0,63




Corrélation : 0,83

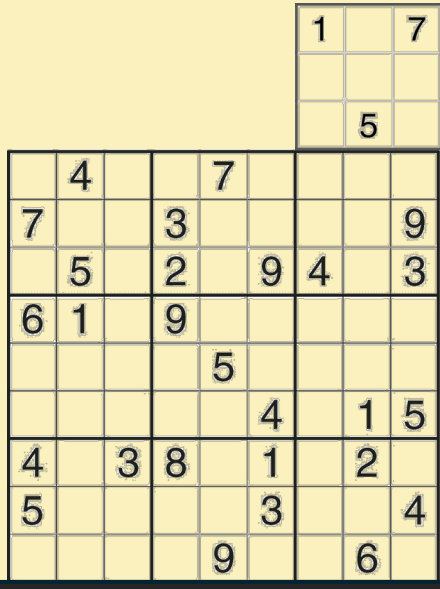
Base évaluée par un autre procédé algorithmique

Base de données 2 : https://huggingface.co/datasets/imone/sudoku-hard-v2

# 3 – Lien avec le problème SAT

Sudoku sans variantes : cas par cas

Objectif : méthode d'évaluation généralisée



# 3.A – Conversion en instance de SAT

Un sudoku peut être transformé en problème CNF-SAT.

$p_{i,j,k} = \text{Vrai si la case } (i,j) \text{ contient } k$

| | | | |
|---|---|---|---|
| 2 3 | 1 | 1 2 | 4 |
| 2 3 | 4 | 1 2 | 1 3 |
| 4 | 2 | 3 | 1 |
| 1 | 3 | 4 | 2 |

$$
\begin{aligned}
& p_{1,2,1} \vee p_{1,3,1} \\
\wedge & \neg p_{1,2,1} \vee \neg p_{1,3,1} \\
\wedge & p_{1,1,2} \vee p_{1,3,2} \\
\wedge & \neg p_{1,1,2} \vee \neg p_{1,3,2} \\
\wedge & p_{1,1,3} \\
\wedge & \ldots
\end{aligned}
$$

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

# 3.B – Résolution

Nouvelle méthode :
Résolution par Quine

| | | | |
|---|---|---|---|
| 3 | 1 | 1 2 | 4 |
| 2 3 | 4 | 1 2 | 1 3 |
| 4 | 2 | 3 | 1 |
| 1 | 3 | 4 | 2 |

$$
\begin{aligned}
& p_{1,2,1} \vee p_{1,3,1} \\
\wedge & \neg p_{1,2,1} \vee \neg p_{1,3,1} \\
\wedge & p_{1,1,2} \vee p_{1,3,2} \\
\wedge & \neg p_{1,1,2} \vee \neg p_{1,3,2} \\
\wedge & p_{1,1,3} \\
\wedge & \ldots \\
\wedge & \neg p_{1,1,3} \vee \ldots
\end{aligned}
$$

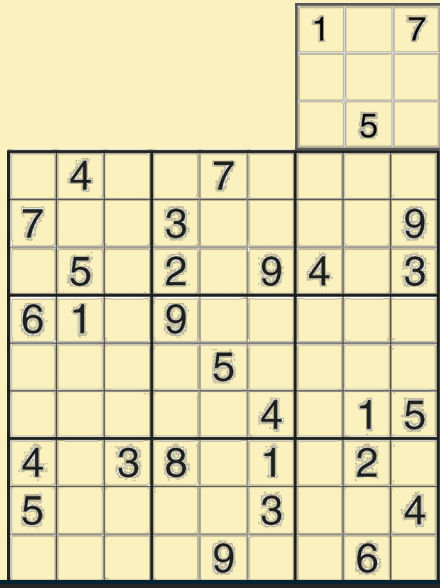
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | 1 | | 7 |
| | | | | | | | | |
| | | | | | | | 5 | |
| | 4 | | | 7 | | | | |
| 7 | | | 3 | | | | | 9 |
| | 5 | | 2 | | 9 | 4 | | 3 |
| 6 | 1 | | 9 | | | | | |
| | | | | 5 | | | | |
| | | | | | 4 | | 1 | 5 |
| 4 | | 3 | 8 | | 1 | |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

# 3.B – Résolution

Nouvelle méthode :
Résolution par Quine, modifié (pseudo-code en annexe)

Prend en compte les deux cas, cherche les points communs.

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

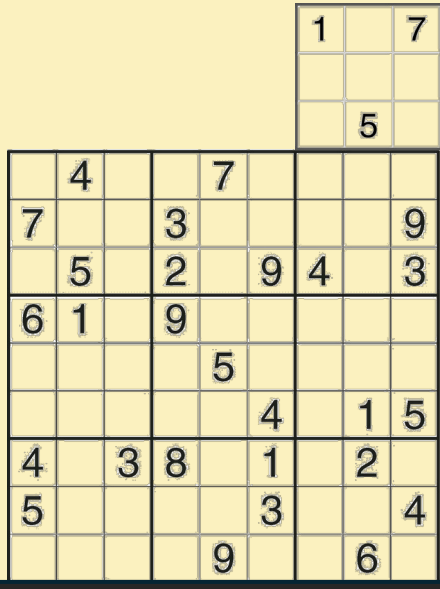


# 3.C – Évaluation

Critères de difficulté :
- Profondeur de l'arbre des disjonctions
- Nombre de disjonctions
- Nombre d'éliminations de variables par Quine

$$d = \alpha(\mathit{profondeur}) + \beta \times n_d + \gamma \times n_Q$$

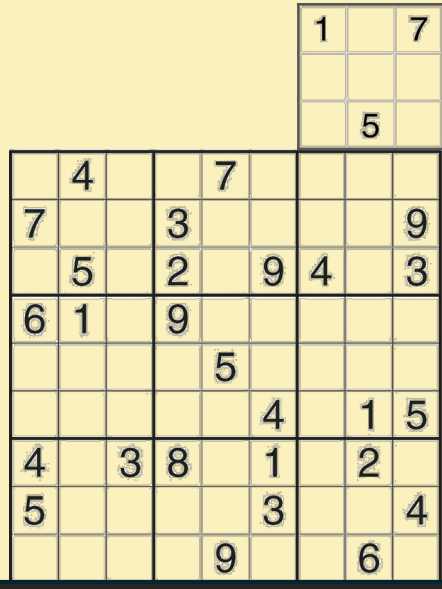
Recherche d'une heuristique de choix de variable pour Quine pour évaluer la difficulté au mieux



# 3.C – Évaluation

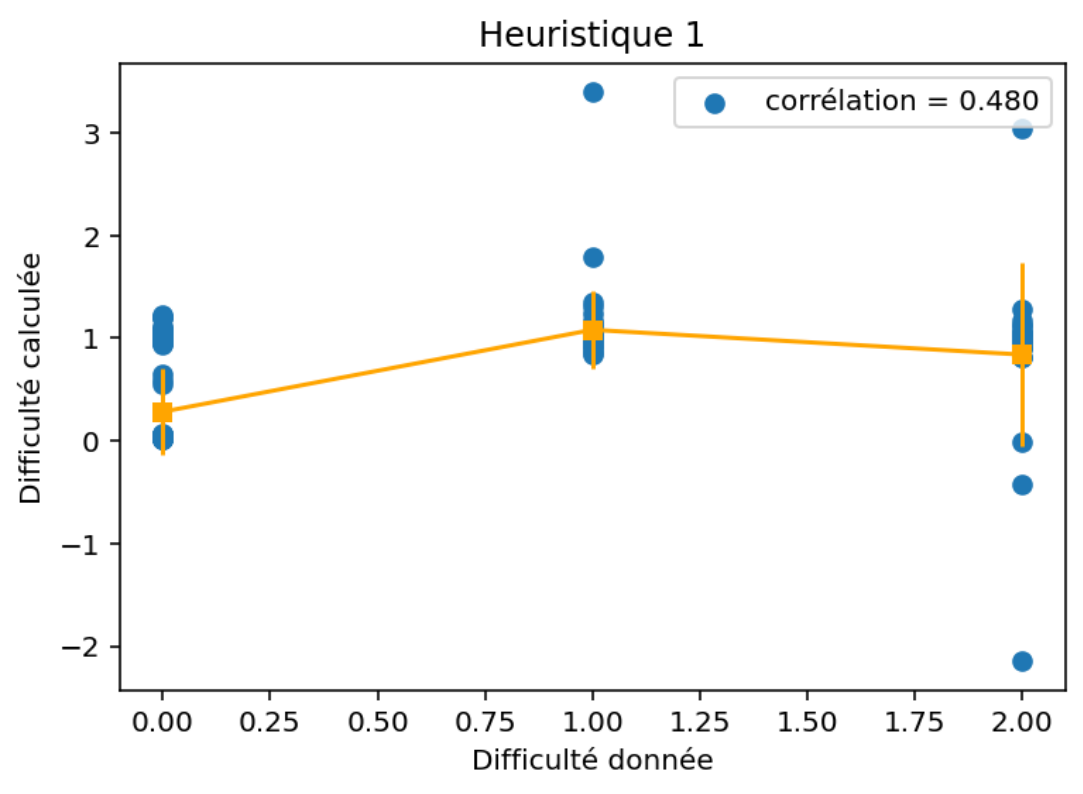
Heuristiques de choix de variable :

1. Choix aléatoire
2. Minimisation de la taille des clauses dont la variable fait partie
3. Maximisation de la taille des clauses dont la variable fait partie



# 3.C – Évaluation - Résultats

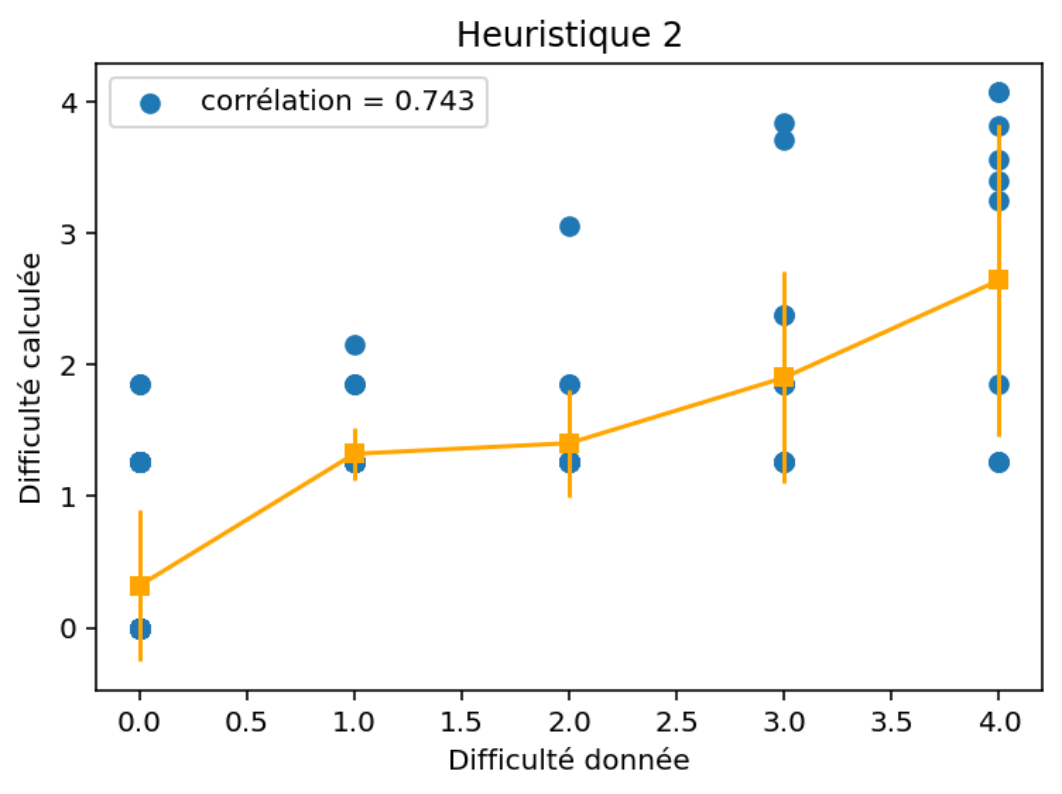
Heuristique 1 :




| | |
|---|---|
| $\alpha(0)$ | 0 ,9 |
| $\alpha(1)$ | 0,7 |
| $\alpha(2)$ | 0,2 |
| $\beta$ | 2 e-2 |
| $\gamma$ | 2 e-4 |

# 3.C – Évaluation - Résultats

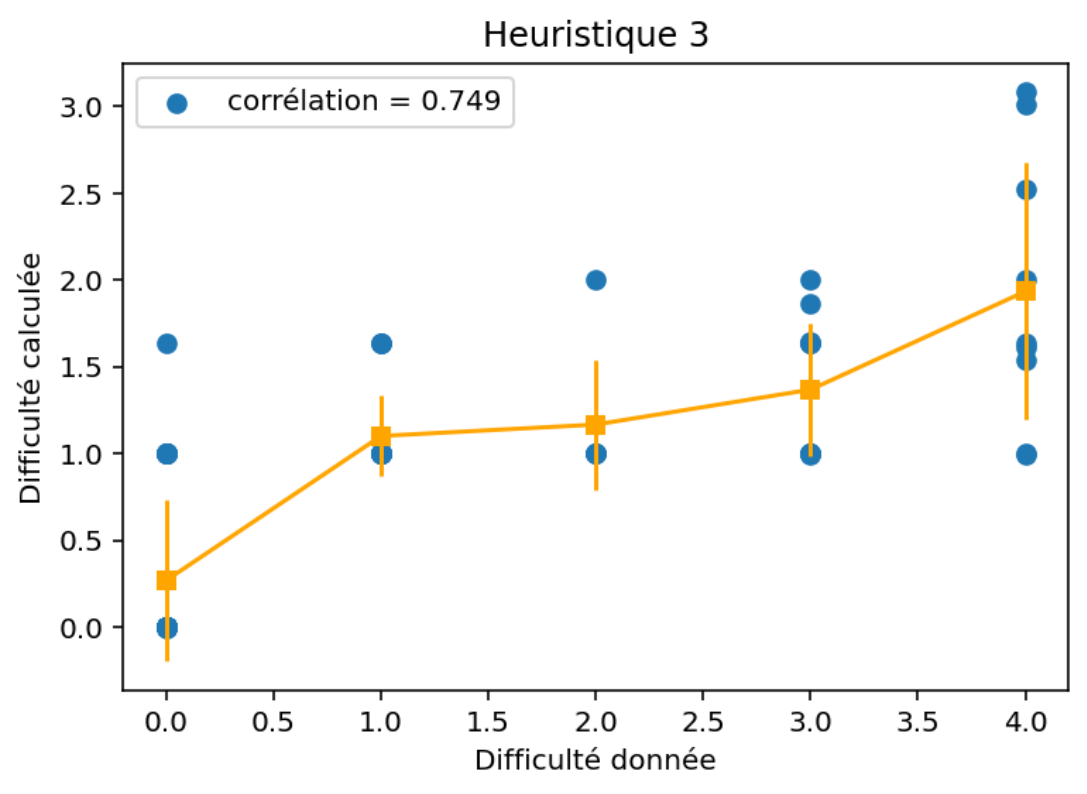
Heuristique 2 :




| | |
|---|---|
| $\alpha(0)$ | 1,3 |
| $\alpha(1)$ | 2,0 |
| $\alpha(2)$ | 2,2 |
| $\beta$ | 9 e-6 |
| $\gamma$ | -2 e-7 |

# 3.C – Évaluation - Résultats

Heuristique 3 :




| | |
|---|---|
| $\alpha(0)$ | 1,1 |
| $\alpha(1)$ | 2,2 |
| $\alpha(2)$ | 2,5 |
| $\beta$ | -2 e-4 |
| $\gamma$ | 3 e-6 |

# Conclusion

Problématique : Pour mesurer la difficulté humaine d'une grille de sudokus, quels algorithmes peut-on élaborer pour une évaluation faite par un ordinateur et quels critères visuels peut-on retenir pour une évaluation faite par un humain ?

TIPE 2025

Félix Landreau- MPI

Fin de l'exposé,

annexes à suivre

# Sommaire des annexes

Annexe 1 : Résultats des autres critères visuels

Annexe 2 : Techniques de résolution

Annexe 3 : Théorème de la projection orthogonale

Annexe 4 : Critères d'évaluation de SAT

Annexe 5 : Pseudo-code de l'algorithme de Quine modifié

Annexe 6 : Références bibliographiques

Annexe 7 : Mon code

# Annexe 1 - Résultats des autres critères

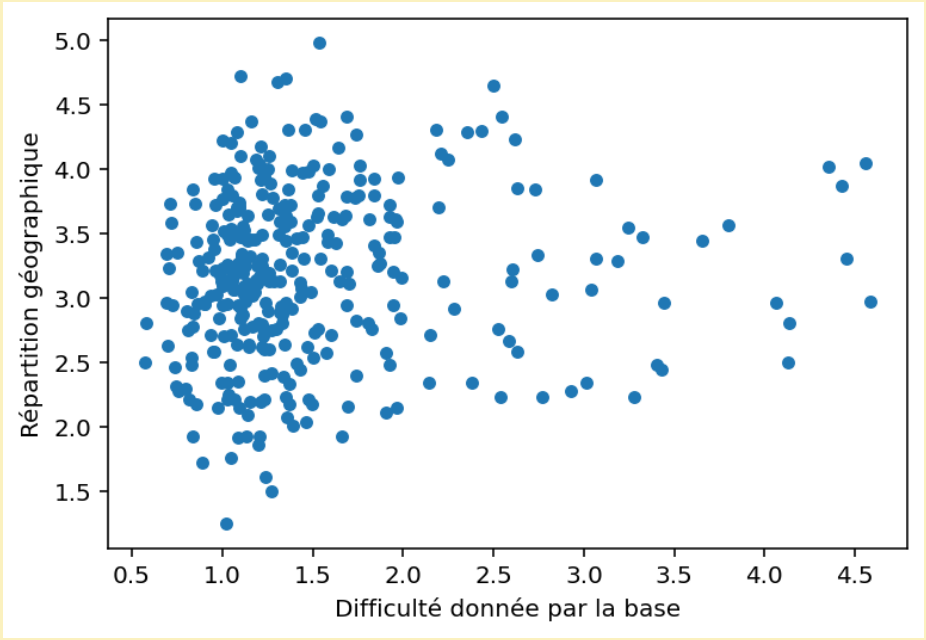
## Répartition géographique des chiffres donnés

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 7 | | 2 | | | | 1 |
| 8 | 4 | | 1 | | | | | |
| | 2 | 1 | 3 | | | | 8 | |
| | | | | 4 | 5 | | | 3 |
| | 6 | | | | | | 4 | |
| 2 | | | 7 | 6 | | | | |
| | 1 | | | | 7 | 3 | 6 | |
| | | | | | 6 | | 7 | 8 |
| 5 | | | | 3 | | 1 | | |

$x_i =$ 3 4 2 3 4 3 2 4 3

$$\sigma = \sqrt{\frac{1}{9}\sum_{i=1}^{9}(x_i - E(x))^2} = 0{,}74$$

## Base de données 1




Corrélation : 0,10

# Annexe 1 – Résultats des autres critères

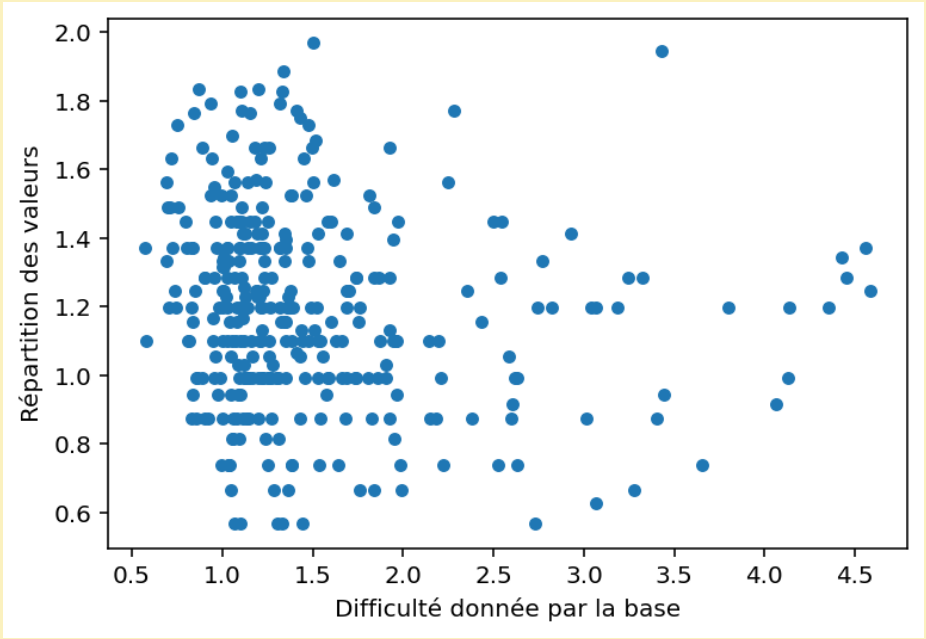
## Répartition entre valeurs des chiffres donnés

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 7 | | 2 | | | | 1 |
| 8 | 4 | | 1 | | | | | |
| | 2 | 1 | 3 | | | | 8 | |
| | | | | 4 | 5 | | | 3 |
| | 6 | | | | | | 4 | |
| 2 | | | 7 | 6 | | | | |
| | 1 | | | | 7 | 3 | 6 | |
| | | | | | 6 | | 7 | 8 |
| 5 | | | | 3 | | 1 | | |

| Valeur | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Nombre | 5 | 3 | 4 | 3 | 2 | 4 | 4 | 3 | 0 |

$$\sigma = \sqrt{\frac{1}{9}\sum_{i=1}^{9}(x_i - E(x))^2} = 1{,}36$$

## Base de données 1




Corrélation : 0,10

# Annexe 2 – Techniques de résolution

Quelques techniques de résolution :

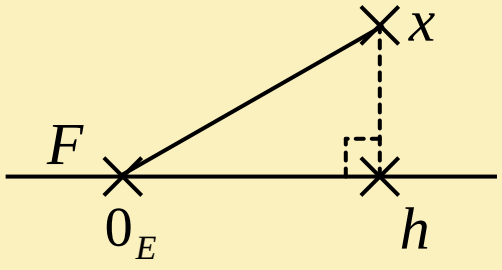
1. Techniques de bas niveau :
    - Dernière case libre / restante
    - Dernier chiffre possible

2. Techniques de plus haut niveau :
    - Notes :
        -> Paire nue / cachée
        -> Triplet ….
    - X-wing – Y Wing , Swordfish
    -

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

# Annexe 3 – Projection orthogonale

Théorème : Soient $E$ un espace préhilbertien, $x \in E$
et $F$ un sous-espace vectoriel de $E$ de dimension finie.
Alors il existe un unique $h \in F$ tel que $x - h \in F^{\perp}$
De plus, $\|x-h\|_2 = \min\{\ \|x-y\|_2\ ,\ y \in F\ \}$




Preuve : Soit $(e_1, e_2, \ldots, e_n)$ une base orthonormée de F

Existence : $h = \sum_{i=1}^{n} \langle x, e_i \rangle e_i$ convient car

$\forall i \in [\![1, n]\!], \langle x-h, e_i \rangle = 0$

Unicité : $h = \sum_{i=1}^{n} \lambda_i e_i \in F$

Donc $\forall i \in [\![1, n]\!], \langle e_i, x-h \rangle = \langle e_i, x \rangle - \lambda = 0$

D'où $h = \sum_{i=1}^{n} \langle x, e_i \rangle e_i$

De plus $\|x-y\|_2^2 = \|x-h\|_2^2 + \|h-y\|_2^2 \geq \|x-h\|_2^2$ car $\langle x-h, h-y \rangle = 0$

# Annexe 3 – Projection orthogonale

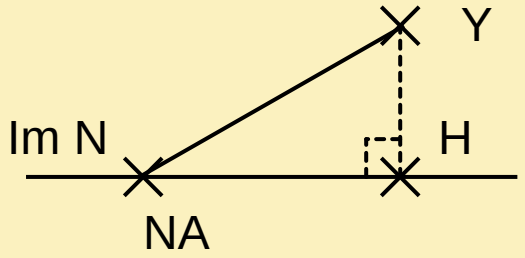
$\|NA - Y\|_2$ est minimal si et seulement si

$NA = H$, donc si et seulement si

$Y - NA \in (\mathrm{Im}\, N)^{\perp}$

Donc les colonnes de N sont des vecteurs orthogonaux à $Y - NA$

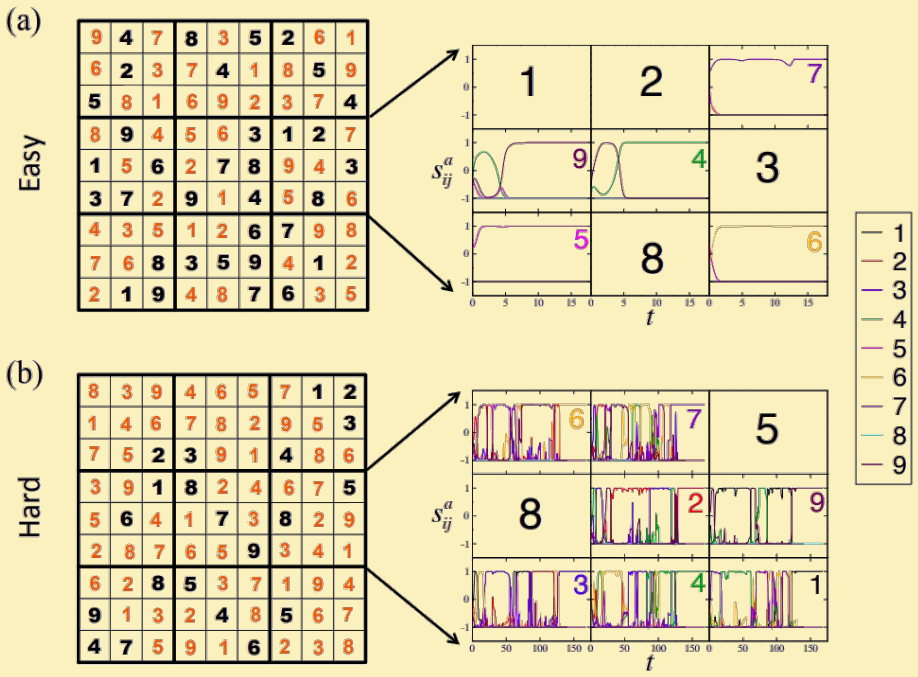
Donc $N^T NA = N^T Y$ (équation normale)



# Annexe 4 – Critères d'évaluation de SAT

- Densité α = nb de clauses / nb de variables

- Durée du chaos :
  Problème continu dynamique
  Chaos déterministe



# Annexe 5 – Algorithme de Quine modifié

**Entrées :** $\varphi$ une CNF
**Sorties :** $S_D$ l'ensemble des valeurs déduites

```
1  v ← h(φ)                    /* Variable sur laquelle sera faite la disjonction. */
2  S_V = {(v, Vrai)}           /* Ensemble des valeurs déduites en supposant v vraie. */
3  φ_V = φ[v ← Vrai]           /* Formule déduite en supposant v vraie. */
4  S_F = {(v, Faux)}
5  φ_F = φ[v ← Faux]
6  tant que S_V ∩ S_F = ∅ faire
7      si une clause de φ_V est vide alors
           /* Alors φ_V est insatisfiable                                    */
8          retourner S_F       /* Les assignations de variables trouvées sont donc
            correctes */
9      fin
10     sinon si une clause de φ_F est vide alors
11         retourner S_V
12     fin
13     sinon si un littéral l est seul dans une clause de φ_V (resp. φ_F) alors
14         Remplacer l par vrai et son complémentaire par faux dans φ_V (resp. φ_F)
15     fin
16     sinon
17         S_V ← Disjonction(φ_V)
18         ou
19         S_F ← Disjonction(φ_F)
20     fin
21 fin
22 retourner S_V ∩ S_F
```

**Algorithme 1 :** Disjonction

# Annexe 5 – Algorithme de Quine modifié

**Entrées :** $\varphi$ une CNF
**Sorties :** $S$ un ensemble de couples (variable,valeur) qui satisfait $\varphi$

```
1 tant que φ est non vide faire
2 |   si un littéral l est seul dans une clause de φ alors
3 |   |   Remplacer l par vrai et son complémentaire par faux dans φ S ← V/F
4 |   sinon
5 |   |   S ← Disjontion(φ)
6 |   fin
7 fin
```

**Algorithme 2 :** Quine Modifié

# Annexe 6 – Références bibliographiques


[1] Radek PELÁNEK. "Difficulty Rating of Sudoku Puzzles : An Overview and Evaluation". In : (2014). DOI : 10.48550/arXiv.1403.7373.

[2] Mária Ercsey-Ravasz & Zoltán TOROCZKAI. "The Chaos Within Sudoku, A Richter-type scale for Sudoku hardness". In : *Sci Rep 2* (2012). DOI : 10.48550/arXiv.1208.0370.

[3] Chungen Xu & Weng XU. "The Model and Algorithm to Estimate the Difficulty Levels of Sudoku Puzzles". In : *CCSE* (2009). DOI : 10.5539/jmr.v1n2p43.

[4] Sheng-Wei WANG. "A Dataset of Sudoku Puzzles With Difficulty Metrics Experienced by Human Players". In : *IEEE Access* 12 (2024), p. 104254-104262. DOI : 10.1109/ACCESS.2024.3434632.

[5] URL : https://huggingface.co/datasets/imone/sudoku-hard-v2.

[6] Will Klieber & Gihwon KWON. "Efficient CNF Encoding for Selecting 1 from N Objec ts". In : *cs.cmu.edu* (2007).

# Annexe

MON_CODE

hiddenPair.c print_content_results.py affiche_notes.c assess_cnf.c assess_nb_clues.c assess_notes.c assess_repartition.c assess_techniques.c backtrack.c boxLineReduction.c calcule_coeffs.c calcule_cout.c consequences_new_number.c consequences_removed_note.c createNotes.c free_grid.c free_notes.c grid_of_string.c print_criteria_results.py print_notes.c simple_colouring.c solve.c solve_cnf.c solve_notes.c solve_simple_notes_backtrack.c sudoku_to_cnf.c swordfish.c updateNotes.c x_wing.c y_wing.c hiddenSingle.c hiddenTriple.c initialize_notes.c lastRemainingCell.c last_free_cell.c last_possible_number.c lecture.c lecture_db_B.c lecture_db_C.c lecture_db_diff.c main.c Makefile moindres_carres.py moindres_carres_log.py nakedPair.c nakedSingle.c nakedTriple.c openness.c pointingPair.c printGrid.c

Dossier mon_code/

Fichier mon_code/hiddenPair.c

```c
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

/*Fonction globale hidden pair
Entrée : une grille de notes
Sortie : booléen (si une nouvelle paire cachée a été trouvée ou non)
Effet de bord : les notes pouvant être retirées grâce à la paire ont été retirées
(Fonction globale appelant successivement hidden pair sur les lignes, les colonnes et les zones )*/
bool hiddenPair_line(bool ***notes);
bool hiddenPair_column(bool ***notes);
bool hiddenPair_zone(bool ***notes);
void free_zones(bool*** zones);

bool hiddenPair(grid_t g) {
   //printf("Coucou from hiddenpair\n");
   bool ok ;
   //printf("On lance sur les lignes\n");
   ok = hiddenPair_line(g->notes);
   if(!ok){
      //printf("On lance sur les colonnes\n");
      ok = hiddenPair_column(g->notes);
   }
   if(!ok){
      //printf("On lance sur les zones\n");
      ok = hiddenPair_zone(g->notes);
   }

   return ok;
}

/*fonction hiddenPair (uniquement sur les lignes )
Entrée : une grille munie de notes
Sortie : un booléen (si une nouvelle paire cachée a été trouvée ou non )*/
bool hiddenPair_line(bool ***notes) {
   for (int i = 0; i < 9; i++) {
      int compteur[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0}; // tableau stockant le nombre d'occurences
                                                     // des valeurs dans les notes (initialisé à
                                                     // zéro à chaque nouvelle ligne étudiée)
      for (int valeur = 0; valeur < 9; valeur++) {
         for (int j = 0; j < 9; j++) {
            if (notes[i][j][valeur]) {
               compteur[valeur]++;
            }
         }
      }
      // le tableau est rempli, on peut donc faire
      // une étude exhaustive des paires possibles
      for (int j = 0; j < 9; j++) {
         if (compteur[j] == 2) {
            for (int k = j + 1; k < 9;
                  k++) { // on recherche une paire parmis les valeurs suivantes (la
                                 // relation de paire étant symetrique)
               if (compteur[k] == 2) { // la seconde valeur est aussi présente deux
                                       // fois, on va donc regarder leurs places
                  // une valeur ne pouvant être présente qu'une seule fois dans une
```

```
                // case, si le nombre total de cases occupées par ces notes
est de
                // deux alors la paire est conforme
                bool place[9] = {
                      false, false, false, false, false,
                      false, false, false, false}; // on utilise ce tableau
pour
                                          // enregistrer les positions des
                                          // paires potentielles
                int count = 0;
                for (int l = 0; l < 9; l++) {
                   if (notes[i][l][k] || notes[i][l][j]) {
                      place[l] = true;
                      count++;
                   }
                }
                if (count == 2) {
                   // la paire est conforme, on peut donc élaguer
                   // une Hidden pair déjà élaguée est identifiée comme une
hidden
                   // pair, on rajoute donc un booléen qui signale lorsque
l'on
                   // élague afin de renvoyer true
                   bool verif = false;
                   for (int n = 0; n < 9; n++) {
                      if (place[n]) {
                         for (int m = 0; m < 9; m++) {
                            if (m != k && m != j && notes[i][n][m]) {
                               verif = true; // on a trouvé une note
différente de la
                               // paire dans une des deux cases,
                               // on la met alors à false
                               notes[i][n][m] = false;
                            }
                         }
                      }
                   }
                   // si verif est faux c'est que la paire est déjà élaguée,
on en
                   // cherche alors une autre dans le tableau;
                   if (verif) {
                      //printf("Technique : hiddenPair ligne\n");
                      //printf("%d et %d forme une paire ligne %d \n", j + 1,
k + 1, i + 1);

                      return true;
                   }
                }
             }
          }
       }
    }
    // tout les lignes on été parcouru
    //printf("Grille parcourue en entier\n");

    return false;
}

/*fonction hiddenPair sur les colonne (même fonctionnement que la fonction
précédente )*/
bool hiddenPair_column(bool ***notes) {
    for (int i = 0; i < 9; i++) {
       int compteur[9] = {0, 0, 0, 0, 0,
                                       0, 0, 0, 0}; // tableau stockant le nombre
d'occurences
                                                    // des valeurs dans les
notes (initialisé à
                                                    // zéro à chaque nouvelle
ligne étudié)
       for (int valeur = 0; valeur < 9; valeur++) {
          for (int j = 0; j < 9; j++) {
             if (notes[j][i][valeur]) {
                compteur[valeur]++;
             }
          }
       }
       // le tableau est rempli, on peut donc faire une étude exhaustive des
pairs
       // possible
       for (int j = 0; j < 9; j++) {
          if (compteur[j] == 2) {
             for (int k = j + 1; k < 9;
                    k++) { // on recherche une paire parmis les valeurs suivante
(la
                                 // relation de paire étant symetrique)
                if (compteur[k] == 2) { // la seconde valeur est aussi présente
deux
                                                    // fois, on va donc regarder
leurs places
                   // une valeur ne pouvant être présente qu'une seule fois dans
une
                   // case, si le nombre totale de case occupé par ces notes est
de
```

```
                // deux alors la paire ets conforme
                bool place[9] = {
                      false, false, false, false, false,
                      false, false, false, false}; // on utilise ce tableau
pour
                                                                  //
enregistrer les positions des
                                                                  // paires
potentielles
                int count = 0;
                for (int l = 0; l < 9; l++) {
                   if (notes[l][i][k] || notes[l][i][j]) {
                      place[l] = true;
                      count++;
                   }
                }
                if (count == 2) {
                   // la paire est conforme, on peut donc élaguer
                   // une Hidden pair déjà élaguée est identifié comme une
hidden
                   // pair, on rajoute donc un booléen qui signale lorsque
l'on
                   // élague afin de renvoyer true
                   bool verif = false;
                   for (int n = 0; n < 9; n++) {
                      if (place[n]) {
                         for (int m = 0; m < 9; m++) {
                            if (m != k && m != j && notes[n][i][m]) {
                               verif = true; // on a trouvé une note
différente de la
                                                    // paire dans une des
deux cases, on la met
                                                    // alors à false
                               notes[n][i][m] = false;
                            }
                         }
                      }
                   }
                   // si verif est faux c'est que la paire est déjà élagué,
on en
                   // cherche alors une autre dans le tableau;
                   if (verif) {
                      //printf("Technique : hiddenPair colonne\n");
                      //printf("%d et %d forme une paire colonne %d \n", j +
1, k + 1, i + 1);
                      return true;
                   }
                }
             }
          }
       }
       // tout les colonnes on été parcouru
       //printf("Grille parcourue en entier\n");
       return false;
    }

    /*fonction hiddenPair sur les zones (même principe mais on convertie la grille
     * pour la parcourir selon les zones)*/
    bool hiddenPair_zone(bool ***notes) {
       // conversion de la grille en zones
       bool ***zones = malloc(9 * sizeof(bool **));
       // on construit une grille répartis en zones :
       // plus simple pour cette étude
       // même morceau de fonction que last_remaining_cell_zone
       assert(zones != NULL);
       for (int i = 0; i < 9; i++) {
          zones[i] = malloc(9 * sizeof(bool **));
          assert(zones[i] != NULL);
          for (int j = 0; j < 9; j++) {
             zones[i][j] = notes[3*(i/3) + j/3][3*(i%3) + j%3] ;
          }
       }



       for (int i = 0; i < 9; i++) {
          int compteur[9] = {0, 0, 0, 0, 0,
                                          0, 0, 0, 0}; // tableau stockant le nombre
d'occurence
                                                                  // des valeurs dans les
notes (initialisé à
                                                                  // zéro à chaque nouvelle
ligne étudié)
          for (int valeur = 0; valeur < 9; valeur++) {
             for (int j = 0; j < 9; j++) {
                if (zones[i][j][valeur]) {
                   compteur[valeur]++;
                }
             }
          }
          // le tableau est rempli, on peut donc faire une étude exhaustive des
pairs
          // possible
```

```
      for (int j = 0; j < 9; j++) {
         if (compteur[j] == 2) {
            for (int k = j + 1; k < 9;
                  k++) { // on recherche une paire parmis les valeurs
suivantes (la
                              // relation de paire étant symetrique)
               if (compteur[k] == 2) { // la seconde valeur est aussi présente
deux
                                                   // fois, on va donc regarder
leurs places
                  // une valeur ne pouvant être présente qu'une seule fois dans
une
                  // case, si le nombre totale de case occupé par ces notes est
de
                  // deux alors la paire ets conforme
                  bool place[9] = {
                        false, false, false, false, false,
                        false, false, false, false}; // on utilise ce tableau
pour
                                                                  //
enregistrer les positions des
                                                                  // paires
potentielles
                  int count = 0;
                  for (int l = 0; l < 9; l++) {
                     if (zones[i][l][k] || zones[i][l][j]) {
                        place[l] = true;
                        count++;
                     }
                  }
                  if (count == 2) {
                     // la paire est conforme, on peut donc élaguer
                     // une Hidden pair déjà élaguée est identifié comme une
hidden
                     // pair, on rajoute donc un booléen qui signale lorsque
l'on
                     // élague afin de renvoyer true
                     bool verif = false;
                     for (int n = 0; n < 9; n++) {
                        if (place[n]) {
                           for (int m = 0; m < 9; m++) {
                              if (m != k && m != j && zones[i][n][m]) {
                                 verif = true; // on a trouvé une note
différente de la
                                                         // paire dans une des
deux cases, on la met
                                                         // alors à false
                                 zones[i][n][m] = false;
                                 notes[3 * (i / 3) + n / 3][3 * (i % 3) + n %
3][m] =
                                       false;
                              }
                           }
                        }
                     }
                     // si verif est faux c'est que la paire est déjà élagué,
on en
                     // cherche alors une autre dans le tableau;
                     if (verif) {
                        //printf("Technique : hiddenPair zone\n");
                        //printf("%d et %d forme une paire zone %d \n", j + 1,
k + 1, i + 1);
                        free_zones(zones);
                        return true;
                     }
                  }
               }
            }
         }
      }
   }
   // tout les zones on été parcouru
   //printf("Grille parcourue en entier\n");
   free_zones(zones);
   return false;
}
```

Fichier mon_code/print_content_results.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon May  5 18:13:53 2025

@author: felix
"""

import matplotlib.pyplot as plt
import numpy as np

f = open("resultats.txt","r")

l = f.read()
m = np.matrix(l)
f.close()
```

```python
#print(m)

dimensions= m.shape
n,p = dimensions

print(n)

coeffs = m[n-2]
print(coeffs)
c = p * [0]
for i in range(p) :
    c[i] = coeffs[0,i]
coeffs =c
print(coeffs)
coeffs_first_use = m[n-1]
c = p * [0]
for i in range(p) :
    c[i] = coeffs_first_use[0,i]
coeffs_first_use =c


n = n-1

diff_calculee = (n-2) * [42]
diff_donnee = (n-2) * [42]
for i in range(n-2) :
    diff = 0
    for j in range(p-1) :
        diff += m[i,j]*coeffs[j]
        if m[i,j] > 0.001 :
            diff += coeffs_first_use[j]
    diff_calculee[i] = diff
    diff_donnee[i] = m[i,p-1]

identite = range(n-2)

#trie les tableaux
t = []
for i in range(n-2):
    t.append((diff_donnee[i],diff_calculee[i]))
t.sort()
for i in range(n-2):
    diff_calculee[i] = t[i][1]
    diff_donnee[i] = t[i][0]
```

```python
#plt.semilogy()
#plt.scatter(identite,diff_calculee,s=40, label="Difficulté calculée")
#plt.scatter(identite,diff_donnee,s=40, label="Difficulté donnée")
plt.xlabel("Difficulté donnée par la base")
plt.ylabel("Difficulté calculée")
plt.scatter(diff_donnee, diff_calculee)
#plt.errorbar(dc_per_dd, means, std)
plt.ylabel("Difficulté calculée")
plt.title("Recuit simulé avec coeffs de première utilisation")
#plt.legend()
plt.show()



######## Calcule la corrélation #######

dcmoy = np.average(diff_calculee)
ddmoy = np.average(diff_donnee)
corr = 0
for i in range(n-2) :
    corr += (diff_calculee[i] - dcmoy)*(diff_donnee[i]-ddmoy)

corr/= (np.std(diff_donnee) * np.std(diff_calculee) * (n-2))

print(corr)
```

Fichier mon_code/affiche_notes.c

```c
/*fonction d'affichage des notes*/

#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void affiche_notes(bool ***notes);

void affiche_notes(bool ***notes) {
  for (int j1 = 0; j1 < 9; j1++) {       // boucle sur les lignes des cases
    for (int j2 = 0; j2 < 3; j2++) {     // boucle sur les lignes des notes
      for (int i1 = 0; i1 < 9; i1++) {   // boucle sur les colonnes des cases
        for (int i2 = 0; i2 < 3; i2++) { // boucle sur les colonnes des notes
          if (notes[j1][i1][3 * j2 + i2]) {
            printf("%d ", 3 * j2 + i2 + 1);
          } else {
            printf("  "); // si la note est à false on print un caractère vide
                           // sinon on print l'indice de la boucle
```

```
          }
        }
        if (i1 == 2 || i1 == 5) {
          printf(" |");
        }
        printf("|  ");
        if (i1 == 8) {
          printf("\n");
        }
      }
    }
    if (j1 == 2 || j1 == 5) {

printf("-------------------------------------------------------------------"
             "------------------\n");
    } else {
      printf("------------------------ || ------------------------ || "
             "-------------------------\n");
    }
  }
}
```

Fichier mon_code/assess_cnf.c

```
#include<stdlib.h>
#include<assert.h>
#include<stdbool.h>
#include<stdio.h>

typedef struct var_s{
    /* Une variable est de la forme p_i,j,k, elle indique si la case i,j
contient k */
    /* 0 <= i,j < 9 */
    /* 1 <= k <= 9 */
    int i;
    int j;
    int k;
}var;

typedef struct {
    /* On représente un litéral par une variable (un entier)
    et une positivite (1 si litéral positif, 0 si litéral négatif)
    On représente donc une clause par un tableau de litéraux */
    int nb_lit ;
    var* vars ;
    bool* positif ;
}clause;

typedef struct {
    /* Ce type est beaucoup moins général ; le filtre est spécifique à cet
usage */
    int nb_var ;
    var* vars ;
    bool* filtre ;
}clause_1in9;


struct k_cnf_s {
    int m ; //nb_clauses
    int k ;
    clause* clauses ;
};
typedef struct k_cnf_s* k_cnf;

void free_clause(clause c);
k_cnf sudoku_to_cnf(int** grid);
void printGrid(int** grid);
void print_k_cnf(k_cnf f);

float assess_cnf(int** grid){
    printGrid(grid);
    k_cnf f = sudoku_to_cnf(grid);
    // on compte les variables
    int n = 0;
    int m = f->m ;
    print_k_cnf(f);
    bool*** existing_variables = malloc(9*sizeof(bool**));
    assert(existing_variables!=NULL);
    for(int i = 0; i<9; i++){
        existing_variables[i] = malloc(9*sizeof(bool*));
        assert(existing_variables[i]!=NULL);
        for(int j = 0; j<9; j++){
            existing_variables[i][j] = malloc(9*sizeof(bool));
            assert(existing_variables[i][j]!=NULL);
            for(int k = 0; k<9; k++){
                existing_variables[i][j][k] = false;
            }
        }
    }
    for(int a = 0; a<f->m; a++){
        for(int b = 0; b<f->clauses[a].nb_lit; b++){
            if(!existing_variables[f->clauses[a].vars[b].i][f-
>clauses[a].vars[b].j][f->clauses[a].vars[b].k]){
                existing_variables[f->clauses[a].vars[b].i][f-
```

```
>clauses[a].vars[b].j][f->clauses[a].vars[b].k] = true ;
                n++ ;
            }
        }
    }
    for(int i = 0; i<9; i++){
        for(int j = 0; j<9; j++){
            free(existing_variables[i][j]);
        }
        free(existing_variables[i]);
    }
    free(existing_variables);
    for(int i = 0; i<m; i++){
        free_clause(f->clauses[i]);
    }
    free(f->clauses);
    free(f);
    assert(n>0);
    return (1.*m)/n;
}
```

Fichier mon_code/assess_nb_clues.c

```
int assess_nb_clues(int** grid){

    int n = 0 ;
    for(int i = 0; i<9; i++){
        for(int j = 0; j<9; j++){
            if(grid[i][j]>0){
                n++ ;
            }
        }
    }
    return n ;
}
```

Fichier mon_code/assess_notes.c

```
#include<stdbool.h>
#include<stdlib.h>
#include<assert.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

bool*** initialize_notes(grid_t g) ;

int assess_nb_notes(int** grid){
    grid_t g = malloc(sizeof(struct grid_s));
    assert(g!=NULL);
    g->grid = grid ;
    initialize_notes(g);

    int count = 0 ;
    for(int i = 0; i<9; i++){
        for(int j = 0; j<9; j++){
            for(int k = 0; k<9; k++){
                if(g->notes[i][j][k]){
                    count++;
                }
            }
        }
    }
    return count ;

}
```

Fichier mon_code/assess_repartition.c

```
#include <math.h>

int assess_nb_clues(int** grid);




float assess_repartition(int** grid){

    float r = 0 ;
    float moy = assess_nb_clues(grid)/9. ;

    float v= 0 ;
    for (int i = 0; i<9; i++){
        // mesure la répartition sur les lignes
        int s = 0;
        for(int j = 0; j<9; j++){
            if(grid[i][j]!=0){
                s++;
            }
        }
        /* A modifier !!!!!!!!!!! */
        v+= (s-moy)*(s-moy);
    }
    r+=sqrt(v/9.);
```

```
    v = 0 ;
    for (int j = 0; j<9; j++){
        // mesure la répartition sur les colonnes
        int s = 0;
        for(int i = 0; i<9; i++){
            if(grid[i][j]!=0){
                s++;
            }
        }
        /* A modifier !!!!!!!!!!!! */
        v+= (s-moy)*(s-moy);
    }
    r+=sqrt(v/9.);

    v = 0;
    for (int z = 0; z<9; z++){
        // mesure la répartition sur les zones
        int s = 0;
        for(int c = 0; c<9; c++){
            int i = 3*(z/3) + c/3 ;
            int j = 3*(z%3) + c%3 ;
            if(grid[i][j]!=0){
                s++;
            }
        }
        /* A modifier !!!!!!!!!!!! */
        v+= (s-moy)*(s-moy);
    }
    r+=sqrt(v/9.);

    return r ;
}


float assess_repartition_valeurs(int** grid){

    float moy = assess_nb_clues(grid)/9. ;

    float v= 0 ;
    for (int k = 0; k<9; k++){
        // mesure la répartition sur les lignes
        int s = 0;
        for(int i = 0; i<9; i++){
            for(int j = 0; j<9; j++){
                if(grid[i][j]==k+1){
                    s++;
                }
            }
        }
        /* A modifier !!!!!!!!!!!! */
        v+= (s-moy)*(s-moy);
    }

    return sqrt(v/9.);
}
```

Fichier mon_code/assess_techniques.c

```
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
#include<stdbool.h>

void solve_simple_notes_backtrack(int** grid, float* nb_tech);
void printGrid(int** grid);

float assess_techniques(int** grid, float* coeffs, float* coeffs_first_use){
    float *nb_tech = malloc(13 * sizeof(float));
    assert(nb_tech!=NULL);
    for (int i = 0; i < 13; i++) {
        nb_tech[i] = 0.;
    }
    printGrid(grid);
    solve_simple_notes_backtrack(grid, nb_tech);
    float score = 0. ;
    for(int i = 0; i<13; i++){
        score += nb_tech[i]*coeffs[i];
        if(nb_tech[i]>0.0001){
            score+=coeffs_first_use[i];
        }
    }
    return score ;
}
```

Fichier mon_code/backtrack.c

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;
```

```c
bool ***createNotes();
void updateNotes(grid_t g, int row, int col);

bool solve_notes(grid_t g, bool(**techniques)(grid_t), int n);
bool est_ok(int** grid);

void printGrid(int **grid);
void print_notes(bool ***notes);
void free_grid(int** grid);
void free_notes(bool*** notes);

bool backtrack(grid_t g, bool(**techniques)(grid_t), int n){
    // Entrées : une grille (peut-etre deja une copie due a un precedent appel
de cette fonction),
    //           ses notes
    // Sorties : false si la grille n'a pas de solutions (impossible sur
l'appel de l'original),
    //           true sinon



    //printf("On passe au Backtracking !\n");

    // on trouve la meilleure case
    int best_row ;
    int best_col ;
    int best_nb_notes = 42;
    for(int i =0; i<9; i++){
        for(int j = 0; j<9; j++){
            if (g->grid[i][j] == 0){
                int nb_notes = 0 ;
                for(int k = 0; k<9; k++){
                    if (g->notes[i][j][k]){
                        nb_notes ++ ;
                    }
                }
                if (nb_notes < best_nb_notes){
                    best_nb_notes = nb_notes ;
                    best_row = i ;
                    best_col = j ;
                }
            }
        }
    }
    //printf("meilleure case : %d, %d avec %d possibilités\n", best_row,
best_col, best_nb_notes);

if (best_nb_notes <= 0){
    // on a trouvé une incohérence, on retourne donc faux
    //printf("une incohérence !\n");
    return false ;
}

//on fait du backtracking

for(int value = 1; value<=9; value++){
    //pour chaque valeur valable
    if(g->notes[best_row][best_col][value-1]){
        //printf("on essaye avec %d\n", value);
        // on duplique la grille
        grid_t newG = malloc(sizeof(struct grid_s));
        assert(newG!=NULL);

        newG->grid = malloc(9*sizeof(int*));
        assert(newG->grid!=NULL);
        for(int i = 0; i<9; i++){
            newG->grid[i] = malloc(9*sizeof(int));
            assert(newG->grid[i]!=NULL);
        }
        for(int i=0; i<9; i++){
            for(int j=0; j<9; j++){
                newG->grid[i][j] = g->grid[i][j];
            }
        }

        newG->notes = createNotes();
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                for(int k = 0; k<9; k++){
                    newG->notes[i][j][k] = g->notes[i][j][k];
                }
            }
        }

        newG->grid[best_row][best_col] = value ;
        updateNotes(newG, best_row, best_col);
        g->nb_techniques[12] ++;

        /* Test : ne pas ajouter les techniques futures */
        newG->nb_techniques = g->nb_techniques ;
        // newG->nb_techniques  = malloc(13*sizeof(float));
        // assert(newG->nb_techniques!=NULL);
        // for(int i = 0; i<13; i++){
        //  newG->nb_techniques[i] =0;
```

```
// }


/*
int* new_nb_techniques = malloc(10*sizeof(int));
assert(new_nb_techniques!=NULL);
for(int i = 0; i<10; i++){
    new_nb_techniques[i] = nb_techniques[i];
}*/

// on essaye de finir la grille
//printGrid(newG->grid);
//print_notes(newG->notes);
bool finished = solve_notes(newG, techniques, n);

if(est_ok(newG->grid)){


    // si on peut finir la grille avec le guess
    if(finished){
        //printf("On a trouve une solution !\n");
        //printGrid(newG->grid);
        //fflush(stdout);
        for(int i=0; i<9; i++){
            for(int j=0; j<9; j++){
                g->grid[i][j] = newG->grid[i][j];
            }
        }
        /*for(int i = 0; i<10; i++){
            nb_techniques[i] = new_nb_techniques[i];
        }*/
        free_grid(newG->grid);
        free_notes(newG->notes);
        free(newG);
        return true ;
    }
    else {
        bool correct = backtrack(newG, techniques, n);
        if (correct){
            //printf("On a trouve une solution !\n");
            for(int i=0; i<9; i++){
                for(int j=0; j<9; j++){
                    g->grid[i][j] = newG->grid[i][j];
                }
            }
            /*
            for(int i = 0; i<10; i++){
                nb_techniques[i] = new_nb_techniques[i];
            }*/
            free_grid(newG->grid);
            free_notes(newG->notes);
            free(newG);
            return true ;
        }
        else{
            //nb_techniques[9] = new_nb_techniques[9];
            g->notes[best_row][best_col][value-1] = false ;
        }
        // si correct est faux, il faut tester les autres
possibilités !
    }

    }
    free_grid(newG->grid);
    free_notes(newG->notes);
    free(newG);
}
    }
    // si on en arrive la, c'est qu'aucune des possibilités ne marchait
    //printf("Aucune des possibilités n'a fonctionné\n");
    return false ;
}
```

Fichier mon_code/boxLineReduction.c

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

/*La technique de box line reduction consiste à éliminer des indices sur une
lignes ou une colonne car ils sont forcément présents sur cette ligne et dans
une autre zone*/

void updateNotes(grid_t g, int row, int col);


bool boxLineReduction(grid_t g){
```

```
    // on parcourt les lignes
    for(int i = 0; i<9; i++){
        // on parcourt les valeurs
        for(int value = 0; value < 9 ; value ++){

            // on parcourt les zones
            int count = 0 ;
            int zone = -1 ;
            for(int z = 0; z<3; z++){
                // si il y a un indice dans la zone on incrémente le nombre de
zone contenant l'indice
                if (g->notes[i][3*z][value]||g->notes[i][3*z+1][value]||g-
>notes[i][3*z+2][value]){
                    count ++ ;
                    zone = z ;
                }
            }
            // si une seule zone contient des indices de même valeur, cela
signifie que ces indices sont tous dans la même zone, la technique peut alors
être appliquée
            //il est inutile de compter le nombre de fois ou l'indice apparaît
puisque cela fonctionne avec un seul indice
            if(count == 1){
                // on modifie le reste de la zone, en enlevant les indices de
la valeur dans le reste de la zone
                assert(zone!=-1);
                bool verif = false;
                for(int k=0; k<9;k++){
                    if((k<3*(i%3) || k > 3*(i%3)+2) && g->notes[3*(i/3)+k/3]
[3*zone+k%3][value]){
                        g->notes[3*(i/3)+k/3][3*zone+k%3][value] = false;
                        //une valeur à élaguer a été trouvé donc on spécifie
que la technique a été utilisé
                        verif = true ;
                    }
                }
                // uniquement si on a modifié quelque chose !
                if(verif){
                    //printf("Technique : boxLineReduction line\n");
                    //printf("ligne = %d, zone = %d, valeur = %d\n", i, zone +
3*(i/3), value+1);
                    return true;
                }
            }
        }
    }
    // Les boucles suivantes sont les mêmes que précédemment, à la différence
prés que l'on itère sur les colonnes au lieu de le faire sur les lignes
    // on parcourt les colonnes
    for(int j = 0; j<9; j++){
        // on parcourt les valeurs
        for(int value = 0; value < 9 ; value ++){

            // on parcourt les zones
            int count = 0 ;
            int zone = -1 ;
            for(int z = 0; z<3; z++){
                // si il y a un indice dans la zone
                if (g->notes[3*z][j][value]||g->notes[3*z+1][j][value]||g-
>notes[3*z+2][j][value]){
                    count ++ ;
                    zone = z ;
                }
            }
            // si une seule zone contient des indices
            if(count == 1){
                assert(zone!=-1);
                // on modifie le reste de la zone
                bool verif = false ;
                for(int k=0; k<9;k++){
                    if(k%3 != j%3 && g->notes[3*zone+k/3][3*(j/3)+k%3][value])
{
                        g->notes[3*zone+k/3][3*(j/3)+k%3][value] = false;
                        verif = true ;
                    }
                }
                // uniquement si on a modifié quelque chose !
                if(verif){
                    //printf("Technique : boxLineReduction column\n");
                    //printf("colonne = %d, zone = %d, valeur = %d\n", j,
3*zone +j/3, value+1);
                    return true;
                }
            }
        }
    }
    return false ;
}
```

Fichier mon_code/calcule_coeffs.c

```
#include<stdio.h>
#include<time.h>
#include<math.h>
#include<assert.h>
#include<stdlib.h>
```

```c
void print_tab_float(float* tab, int size);
float calcule_cout(float* coeffs, float* coeffs_first_use, float** results,
float* difficulties, int results_size);


float* copy(float* tab, int size){
    float* t = malloc(size*sizeof(float));
    assert(t!=NULL);
    for(int i = 0; i<size; i++){
        t[i] = tab[i];
    }
    return t;
}

void calcule_coeffs(float* coeffs, float* coeffs_first_use, float** results,
float* difficulties, int results_size){
    /* Calcule les valeurs des tableaux coeffs et coeffs_first_use minimisant
l'écart au difficultés, */
    /* selon les moindres carrés */
    /* Ce calcul se fera à l'aide d'un recuit simulé */
    /* https://en.wikipedia.org/wiki/Simulated_annealing */
    //srand(time(NULL));
    float cout = calcule_cout(coeffs, coeffs_first_use, results, difficulties,
results_size);
    float* best_coeffs = copy(coeffs, 13);
    float* best_coeffs_first_use = copy(coeffs_first_use, 13);
    float best_cout = cout ;
    int i = 0;
    float T = 10. ;
    while (T > 0.000001) {
        float* nouveaux_coeffs = copy(coeffs, 13);
        float* nouveaux_coeffs_first_use = copy(coeffs_first_use,13);

        for (int j = 0; j <13; j++) {
            nouveaux_coeffs[j] *= expf((-10 + (rand() % 21))/1000.);
            nouveaux_coeffs_first_use[j] *= expf((-10 + (rand() % 21))/1000.);
        }
        float nouveau_cout = calcule_cout(nouveaux_coeffs,
nouveaux_coeffs_first_use, results, difficulties, results_size);
        if(nouveau_cout<cout || (((rand() % 1000000)/1000000.) < expf((cout-
nouveau_cout)/T))){
            cout = nouveau_cout ;
            for(int i = 0; i<13; i++){
                coeffs[i] = nouveaux_coeffs[i];
                coeffs_first_use[i] = nouveaux_coeffs_first_use[i];
            }
            if(best_cout>cout){
                for(int i = 0; i<13; i++){
                    best_coeffs[i] = coeffs[i] ;
                    best_coeffs_first_use[i] = coeffs_first_use[i] ;
                }
                best_cout = cout ;
            }

        }

        free(nouveaux_coeffs);
        free(nouveaux_coeffs_first_use);

        T = T*0.9999 ;

        if(i%10000 == 0){
            printf("i = %d : \n", i);
            print_tab_float(coeffs, 13);
            print_tab_float(coeffs_first_use, 13);
            printf("Cout = %f\n", cout);
            float ec = sqrt(cout);
            printf("ecart-type = %f\n", ec);
            printf("T = %f\n", T);

        }
        i++;
    }
    print_tab_float(best_coeffs, 13);
    print_tab_float(best_coeffs_first_use, 13);
    //printf("Cout = %f\n", best_cout);
    float ec = sqrt(best_cout);
    //printf("ecart-type = %f\n", ec);
    //printf("T = %f\n", T);

    for(int i = 0; i<13; i++){
        coeffs[i] = best_coeffs[i] ;
        coeffs_first_use[i] = best_coeffs_first_use[i] ;
    }
    free(best_coeffs);
    free(best_coeffs_first_use);


}

void calcule_coeffs_neg(float* coeffs, float* coeffs_first_use, float**
results, float* difficulties, int results_size){
    /* Calcule les valeurs des tableaux coeffs et coeffs_first_use minimisant
```

```
l'écart au difficultés, */
    /* selon les moindres carrés */
    /* Ce calcul se fera à l'aide d'un recuit simulé */
    /* https://en.wikipedia.org/wiki/Simulated_annealing */
    //srand(time(NULL));
    float cout = calcule_cout(coeffs, coeffs_first_use, results, difficulties,
results_size);
    float* best_coeffs = copy(coeffs, 13);
    float* best_coeffs_first_use = copy(coeffs_first_use, 13);
    float best_cout = cout ;
    int i = 0;
    float T = 10. ;
    while (T > 0.000001) {
        float* nouveaux_coeffs = copy(coeffs, 13);
        float* nouveaux_coeffs_first_use = copy(coeffs_first_use,13);

        for (int j = 0; j <13; j++) {
            nouveaux_coeffs[j] += (-10 + (rand() % 21))/1000.;
            //nouveaux_coeffs_first_use[j] += (-10 + (rand() % 21))/1000.;
        }
        float nouveau_cout = calcule_cout(nouveaux_coeffs,
nouveaux_coeffs_first_use, results, difficulties, results_size);
        if(nouveau_cout<cout || (((rand() % 1000000)/1000000.) < expf((cout-
nouveau_cout)/T))){
            cout = nouveau_cout ;
            for(int i = 0; i<13; i++){
                coeffs[i] = nouveaux_coeffs[i];
                coeffs_first_use[i] = nouveaux_coeffs_first_use[i];
            }
            if(best_cout>cout){
                for(int i = 0; i<13; i++){
                    best_coeffs[i] = coeffs[i] ;
                    best_coeffs_first_use[i] = coeffs_first_use[i] ;
                }
                best_cout = cout ;
            }

        }

        free(nouveaux_coeffs);
        free(nouveaux_coeffs_first_use);

        T = T*0.9999 ;

        if(i%10000 == 0){
            printf("i = %d : \n", i);
            print_tab_float(coeffs, 13);
            print_tab_float(coeffs_first_use, 13);
            printf("Cout = %f\n", cout);
            float ec = sqrt(cout);
            printf("ecart-type = %f\n", ec);
            printf("T = %f\n", T);

        }
        i++;
    }
    print_tab_float(best_coeffs, 13);
    print_tab_float(best_coeffs_first_use, 13);
    //printf("Cout = %f\n", best_cout);
    float ec = sqrt(best_cout);
    //printf("ecart-type = %f\n", ec);
    //printf("T = %f\n", T);

    for(int i = 0; i<13; i++){
        coeffs[i] = best_coeffs[i] ;
        coeffs_first_use[i] = best_coeffs_first_use[i] ;
    }
    free(best_coeffs);
    free(best_coeffs_first_use);


}

// void calcule_coeffsv1(float* coeffs, float* coeffs_first_use, float**
results, float* difficulties, int results_size){
//     /* Calcule les valeurs des tableaux coeffs et coeffs_first_use
minimisant l'écart au difficultés, */
//     /* selon les moindres carrés */
//     /* Ce calcul se fera à l'aide d'une recherche linéaire */

//     float cout = calcule_cout(coeffs, coeffs_first_use, results,
difficulties, results_size);
//  float ancien_cout = cout + 42;
//  int i = 0;
//  while (ancien_cout > cout) {
```

```
//      ancien_cout = cout ;
//      for (int j = 0; j <13; j++) {
//          //for (int k = 0; k < 100; k++) {
//              coeffs[j] *= 0.99;
//              float nouveau_cout = calcule_cout(coeffs, coeffs_first_use,
results, difficulties, results_size);
//              if (nouveau_cout >= cout) {
//                  coeffs[j] /= 0.99;
//              } else {
//                  cout = nouveau_cout;
//              }
//              coeffs[j] *= 1.01;
//              nouveau_cout = calcule_cout(coeffs, coeffs_first_use, results,
difficulties, results_size);
//              if (nouveau_cout >= cout) {
//                  coeffs[j] /= 1.01;
//              } else {
//                  cout = nouveau_cout;
//              }
//              coeffs_first_use[j] *= 0.99;
//              nouveau_cout = calcule_cout(coeffs, coeffs_first_use, results,
difficulties, results_size);
//              if (nouveau_cout >= cout) {
//                  coeffs_first_use[j] /= 0.99;
//              } else {
//                  cout = nouveau_cout;
//              }
//              coeffs_first_use[j] *= 1.01;
//              nouveau_cout = calcule_cout(coeffs, coeffs_first_use, results,
difficulties, results_size);
//              if (nouveau_cout >= cout) {
//                  coeffs_first_use[j] /= 1.01;
//              } else {
//                  cout = nouveau_cout;
//              }
//          //}
//      }
//      if(i%10 == 0){
//          //printf("i = %d : \n", i);
//          print_tab_float(coeffs, 13);
//          print_tab_float(coeffs_first_use, 13);
//          //printf("Cout = %f\n", cout);
//          float ec    = sqrt(cout);
//          //printf("ecart-type = %f\n", ec);
//          fflush(stdout);
//      }
//      i++;
//  }

// }
```

Fichier mon_code/calcule_cout.c

```
#include <assert.h>
#include <stdbool.h>



float calcule_cout(float* coeffs, float* coeffs_first_use, float** results,
float* difficulties, int results_size){
    float cout = 0 ;
    for(int i = 0; i<results_size; i++){
        float diff_i = 0 ;
        for(int j = 0; j<13; j++){
            diff_i += results[i][j] * coeffs[j] ;
            if (results[i][j] > 0.001){
                diff_i += coeffs_first_use[j];
            }
        }
        cout += (diff_i-difficulties[i]) * (diff_i-difficulties[i]) ;
    }
    cout = cout /(1. * results_size) ;
    return cout ;
}
```

Fichier mon_code/consequences_new_number.c

```
// #include <assert.h>
// #include <stdbool.h>
// #include <stdio.h>
// #include <stdlib.h>
// #include <time.h>

// typedef struct grid_s {
//  int** grid ;
//  bool*** notes ;
//  float* nb_techniques;
// }grid_s ;

// bool nakedSingle_one_cell(int **grid, bool ***notes, int i, int j);
// float* consequences_removed_note(int **grid, bool ***notes, int i, int j,
int k, float* nb_techniques);
// void updateNotes(grid_t g, int row, int col);
// void printGrid(int** grid);
// void free_notes(bool*** notes);



// bool lastFreeCell_one_cell(grid_t g, int i, int j);
```

```
// void add_float_tabs(float* tab1, float* tab2, int size){
//     for(int i = 0; i<size; i++){
//         tab1[i] += tab2[i];
//     }
// }

// void consequences_new_number(grid_t g, int i, int j) {

//     float* modifs = malloc(13 * sizeof(float));
//     assert(modifs!=NULL);
//     for(int t = 0; t<13; t++){
//         modifs[t] = 0.;
//     }

//     //printf("Consequences new number : i = %d, j = %d, value = %d\n", i, j,
grid[i][j]);
//     printGrid(grid);

//     if(lastFreeCell_one_cell(g, i, j)){
//         g->nb_techniques[0] += 2./9. ;
//     }
//     //printf("Coucou\n");
//     int nb_notes_modified = 0;
//     bool*** newNotes = malloc(9*sizeof(bool**));
//     assert(newNotes!=NULL);
//     for(int i = 0; i<9; i++){
//         newNotes[i] = malloc(9*sizeof(bool*));
//         assert(newNotes[i]!=NULL);
//         for(int j = 0; j<9; j++){
//             newNotes[i][j] = malloc(9*sizeof(bool));
//             assert(newNotes[i][j] != NULL);
//             for(int k = 0; k<9; k++){
//                 newNotes[i][j][k] = g->notes[i][j][k] ;
//             }
//         }
//     }
//     updateNotes(g, i, j);
//     // on duplique la grille afin de garder en mémoire les modifications de
notes

//     // ce nouveau tableau de techniques comptabilise les "nouveaux" gains
//     // voir a la fin de la focntion pour l'utilite

//     // nakedSingle
//     for (int i1 = 0; i1 < 9; i1++) {
//         // Alternative :
//         if(g->notes[i1][j][g->grid[i][j]-1] != newNotes[i1][j][grid[i][j]-1]
&& i1!=i){
//             float* to_add = consequences_removed_note(grid, newNotes, i1, j,
grid[i][j]-1, nb_techniques);
//             add_float_tabs(modifs, to_add, 13) ;
//             nb_notes_modified ++;
//             free(to_add);
//         }
//     }
//     for (int j1 = 0; j1 < 9; j1++) {
//         if(notes[i][j1][grid[i][j]-1] != newNotes[i][j1][grid[i][j]-1]&&
j1 != j){
//             float* to_add = consequences_removed_note(grid, newNotes, i, j1,
grid[i][j]-1, nb_techniques);
//             add_float_tabs(modifs, to_add, 13) ;
//             nb_notes_modified ++;
//             free(to_add);
//         }
//     }
//     for (int i1 = 0; i1 < 3; i1++) {
//         for (int j1 = 0; j1 < 3; j1++) {
//             if(notes[i1][j1][grid[i][j]-1] != newNotes[i1][j1][grid[i][j]-1]
&& (i1 != i || j1 != j)){
//                 float* to_add = consequences_removed_note(grid, newNotes,
i1, j1, grid[i][j]-1, nb_techniques);
//                 add_float_tabs(modifs, to_add, 13) ;
//                 nb_notes_modified ++;
//                 free(to_add);
//             }
//         }
//     }

//     for(int k = 0; k<9; k++){
//         if(notes[i][j][k] != newNotes[i][j][k] && k+1 != grid[i][j]){
//             float* to_add = consequences_removed_note(grid, newNotes, i, j,
k, nb_techniques);
//             add_float_tabs(modifs, to_add, 13) ;
//             free(to_add);
//             nb_notes_modified ++;
//         }
//     }

//     for(int i1 = 0; i1<9; i1++){
//         for(int j1 = 0; j1<9; j1++){
//             for(int k1 = 0; k1<9; k1++){
//                 notes[i1][j1][k1] = newNotes[i1][j1][k1];
//             }
```

```
//         }
//     }
//     free_notes(newNotes);

//     for(int t = 0; t<13; t++){
//         nb_techniques[t] += modifs[t] * nb_notes_modified ;
//     }
//     free(modifs);
//     // On a du faire nb_notes_modified essais pour trouver les techniques !
//     // On multiplie donc la difficulté pas nb_notes_modified
// }
```

Fichier mon_code/consequences_removed_note.c

```
// #include <assert.h>
// #include <stdbool.h>
// #include <stdio.h>
// #include <stdlib.h>
// #include <time.h>

// typedef struct grid_s {
//  int** grid ;
//  bool*** notes ;
//  float* nb_techniques;
// }grid_s ;

// bool nakedSingle_one_cell(int** grid, bool*** notes, int i, int j, float*
nb_tech);

// bool hiddenSingle_one_cell(int** grid, bool*** notes, int i, int j, int k,
float* nb_tech);

// bool nakedPair_one_cell_value(int** grid, bool ***notes, int i, int j, int
k, float* nb_tech);

// bool pointingPair_one_zone_one_value(bool*** notes, int z, int value);
// bool lastFreeCell_one_cell(int** grid, bool*** notes, int i, int j, float*
nb_tech);

// float* consequences_removed_note(grid_t g, int i, int j, int k){

//     float* modifs = malloc(13*sizeof(float));
//     assert(modifs!=NULL);
//     for(int t = 0; t<13; t++){
//         modifs[t] = 0;
//     }

//     //printf("consequence_removed_note : i = %d , j = %d, value = %d\n", i,
j, k+1);
//     // nakedSingle
//     //printf("On teste le nakedSingle\n");

//     if(g->nb_techniques[2]>0.01){
//         int nb_pos = hiddenSingle_one_cell(grid, notes, i, j, k,
nb_techniques);
//         //nb_tech[1] ++;
//         modifs[2] += nb_pos/9. * 1./81. ;
//     }
//     if(g->nb_techniques[1] > 0.01 && nakedSingle_one_cell(grid, notes, i, j,
nb_techniques)){
//         modifs[1] += 1./81. ;

//     }
//     //printf("On teste le hiddenSingle\n");
//     if(g->nb_techniques[3] > 0.01 && nakedPair_one_cell_value(grid, notes,
i, j, k, nb_techniques)){
//         modifs[3] += 1./81. ;
//     }

//     if(g->nb_techniques[5] > 0.01 && pointingPair_one_zone_one_value(notes,
3*(i/3) + j/3, k)){
//          modifs[5] += 1./81. ;
//     }
//     return modifs ;
// }
```

Fichier mon_code/createNotes.c

```c
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/*fonction create note
Entrée : aucune
Sortie : un tableau de booléen de dimension 3
A la case i,j on trouve un tableau de neuf booléen, spécifiant si l'indice k
peut être posé en i,j*/
bool*** createNotes() {
    //création du tableau dans le tas
    bool*** notes = malloc(9 * sizeof(bool **));
    assert(notes!=NULL);
    for (int i = 0; i < 9; i++) {
        notes[i] = malloc(9 * sizeof(bool *));
```

```
        assert(notes[i]!=NULL);
        for (int j = 0; j < 9; j++) {
            notes[i][j] = malloc(9 * sizeof(bool));
            assert(notes[i][j]!=NULL);
            for (int k = 0; k < 9; k++) {
                notes[i][j][k] = true;
                //on initialise toute la grille à true, la grille sera modifiée
en fonction des indices ultérieurement
            }
        }
    }
    return notes;
}
```

Fichier mon_code/free_grid.c

```
#include <stdlib.h>


void free_grid(int** grid){
    for(int i = 0; i<9; i++){
        free(grid[i]);
    }
    free(grid);
}
```

Fichier mon_code/free_notes.c

```
#include <stdlib.h>
#include <stdbool.h>

void free_notes(bool*** notes){
    for(int i = 0; i<9; i++){
        for(int j = 0; j<9; j++){
            free(notes[i][j]);
        }
        free(notes[i]);
    }
    free(notes);
}
```

Fichier mon_code/grid_of_string.c

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>


int** grid_of_string(char* s){
    int** grid = malloc(9*sizeof(int*));
    assert(grid!=NULL);
    for(int i =0; i<9; i++){
        grid[i] = malloc(9*sizeof(grid));
        assert(grid!=NULL);
        for(int j = 0; j<9; j++){
            grid[i][j] = s[9*i + j] - '0';
        }
    }
    return grid ;
}
```

Fichier mon_code/print_criteria_results.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon May  5 18:13:53 2025

@author: felix
"""

import matplotlib.pyplot as plt
import numpy as np

f = open("results_db_0/results_criteria.txt","r")

l = f.read()
m = np.matrix(l)
f.close()

#print(m)

dimensions= m.shape
n,p = dimensions




n = n - 10

#trie les tableaux
t = []
for i in range(n):
    t.append((m[i,0],m[i,1],m[i,2],m[i,3],m[i,4],m[i,5], m[i,6]))
t.sort()

# t = t[:n-200]
# n = n-200
```

```
diff_donnee = n*[42]
diff_calculee = n * [42]
density = n*[42]
nombre_indices = n * [42]
nb_notes = n*[42]
repartition = n*[42]
repartition_valeurs = n*[42]
mix = n * [42]

for i in range(n):
    diff_donnee[i] = t[i][0]
    diff_calculee[i] = t[i][1]
    density[i] = t[i][2]
    nombre_indices[i] = int( t[i][3])
    nb_notes[i] = t[i][4]
    repartition[i] = t[i][5]
    repartition_valeurs[i] = t[i][6]
    mix[i] = -1.*repartition_valeurs[i] + 1.5*repartition[i]  -
0.8*nombre_indices[i] + 20  # + nb_notes[i] /75



diff_per_ni = []
for i in range(9) :
    diff_per_ni.append([])
means = 9 * [42]
std = 9 * [42]
for i in range(n):
    diff_per_ni[nombre_indices[i]-24].append( diff_donnee[i])

for i in range(9) :
    means[i] = np.average(diff_per_ni[i])
    std[i] = np.std(diff_per_ni[i])




print("n=",n)
identite = range(n)

#plt.semilogy()
#plt.scatter(identite,diff_calculee, label="Difficulté calculée")
#plt.scatter(identite, diff_donnee, s=20, label="Difficulté donnée")
#plt.scatter(identite, density,s=20, label="Densité")
#plt.scatter(diff_donnee,nombre_indices, s=20, label="Difficulté donnée")
#plt.scatter(range(24,33), means, s = 60,label = "Difficulté moyenne")
#plt.errorbar(means,range(24,33),xerr= std, linestyle='None', marker ='s', mfc
= 'orange', mec = "orange", ecolor = "orange", label = "Moyennes et écarts-
types")
#plt.scatter(diff_donnee, nb_notes, s=20, label="Nombre de candidats")
#plt.scatter(diff_donnee, repartition, s=20, label="Répartition géographique")
plt.scatter(diff_donnee, repartition_valeurs, s=20, label="Répartition des
valeurs")
#plt.scatter(identite, mix, s=20, label ="Mix")
plt.ylabel("Répartition des valeurs")
plt.xlabel("Difficulté donnée par la base")
#plt.title("Recuit simulé avec coeffs de première utilisation")
#plt.legend()
plt.show()



######## Calcule la corrélation #######

ddmoy = np.average(diff_donnee)
dcmoy = np.average(diff_calculee)
densitymoy = np.average(density)
nimoy = np.average(nombre_indices)
nbnotesmoy = np.average(nb_notes)
rmoy = np.average(repartition)
rvmoy = np.average(repartition_valeurs)
mmoy = np.average(mix)

corr_01 = 0
corr_02 = 0
corr_12 = 0
corr_03 = 0
corr_04 = 0
corr_05 = 0
corr_35 = 0
corr_06 = 0
corr_07 = 0
for i in range(n) :
    corr_01 += (diff_calculee[i] - dcmoy)*(diff_donnee[i]-ddmoy)
    corr_02 += (density[i] - densitymoy)*(diff_donnee[i]-ddmoy)
    corr_03 += (diff_donnee[i] - ddmoy)*(nombre_indices[i]-nimoy)
    corr_04 += (nb_notes[i] - nbnotesmoy)*(diff_donnee[i]-ddmoy)
    corr_12 += (nombre_indices[i] - nimoy)*(diff_calculee[i]-dcmoy)
    corr_05 += (repartition[i] - rmoy)*(diff_donnee[i]-ddmoy)
    corr_35 += (nombre_indices[i]-nimoy)*(repartition[i]-rmoy)
    corr_06 += (diff_donnee[i]-ddmoy)*(repartition_valeurs[i]-rvmoy)
    corr_07 += (diff_donnee[i]-ddmoy)*(mix[i]-mmoy)
```

```
corr_01/= (np.std(diff_donnee) * np.std(diff_calculee) * n)
corr_02/= (np.std(diff_donnee) * np.std(density) * n)
corr_03/= (np.std(diff_donnee) * np.std(nombre_indices) * n)
corr_04/= (np.std(diff_donnee) * np.std(nb_notes) * n)
corr_05/= (np.std(diff_donnee) * np.std(repartition) * n)
corr_12/= (np.std(nombre_indices) * np.std(diff_calculee) * n)
corr_35/= (np.std(repartition)*np.std(nombre_indices)*n)
corr_06/= (np.std(repartition_valeurs)*np.std(diff_donnee)*n)
corr_07/= (np.std(mix)*np.std(diff_donnee)*n)

print("corr_01 =", corr_01)
print("corr_02 =", corr_02)
print("corr_03 =", corr_03)
print("corr_04 =", corr_04)
print("corr_05 =", corr_05)
print("corr_12 =", corr_12)
print("corr_35 = ",corr_35)
print("corr_06 = ", corr_06)
print("corr_07 = ", corr_07)
```

Fichier mon_code/print_notes.c

```
/*fonction d'affichage des notes*/

#include <assert.h>
#include <stdbool.h>
#include <stdio.h>


void print_notes(bool ***notes);

void print_notes(bool ***notes) {
  for (int j1 = 0; j1 < 9; j1++) {       // boucle sur les lignes des cases
    for (int j2 = 0; j2 < 3; j2++) {     // boucle sur les lignes des notes
      for (int i1 = 0; i1 < 9; i1++) {   // boucle sur les colonnes des cases
        for (int i2 = 0; i2 < 3; i2++) { // boucle sur les colonnes des notes
          if (notes[j1][i1][3 * j2 + i2]) {
            printf("%d ", 3 * j2 + i2 + 1);
          } else {
            printf("  "); // si la note est à false on print un caractère vide
                          // sinon on print l'indice de la boucle
          }
        }
        if (i1 == 2 || i1 == 5) {
          printf(" |");
        }
        printf("|  ");
        if (i1 == 8) {
          printf("\n");
        }
      }
    }
    if (j1 == 2 || j1 == 5) {

printf("-------------------------------------------------------------------------------
\n");
    } else {
      printf("----------------------- || ----------------------- ||
-------------------------\n");
    }
  }
}
```

Fichier mon_code/simple_colouring.c

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <assert.h>

// remarque : c'est rigolo, ça utilise les graphes bipartis

/*
typedef struct chaine {
    int i1 ;
    int j1;
    int i2;
    int j2;
}chaine;
*/
void parcours(int i, int j, bool* is_a_chain_line, bool* is_a_chain_column,
    bool* is_a_chain_zone, int* ligne1, int* ligne2, int* colonne1, int*
colonne2,
    int* lignez1, int* lignez2, int* colonnez1, int* colonnez2, int** colour);

bool simple_colouring(bool*** notes){

    for(int n = 0; n<9; n++){

        // Première partie : recherche des chaines
        //similaire au x-wing
        // essai
        //chaine* ligne[9] = {NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL};

        // indices de départ de l'étape 2
        int i_dep = -1 ;
```

```
        int j_dep = -1 ;

        // indices de deux colonnes contenat des indices (utile si count[i] =
2)
        bool is_a_chain_line[9] =
{false,false,false,false,false,false,false,false,false};
        int colonne1[9] = {-1, -1, -1, -1, -1, -1, -1, -1, -1};
        int colonne2[9] = {-1, -1, -1, -1, -1, -1, -1, -1, -1};
        for(int i = 0; i<9; i++){
            int count = 0;
            for(int j = 0; j<9; j++){
                if(notes[i][j][n]){
                    count++ ;
                    if(colonne1[i]==-1){
                        colonne1[i] = j;
                    }
                    else{
                        colonne2[i] = j;
                    }
                }
            }
            if (count == 2){
                /*chaine* c = malloc(sizeof(chaine));
                c-> i1 = i;
                c->i2 = i;
                c->j1 = colonne2;
                ligne[i] =
                */
                i_dep = i;
                j_dep = colonne1[i];
                is_a_chain_line[i] = true;
            }
        }

        bool is_a_chain_column[9] =
{false,false,false,false,false,false,false,false,false};
        int ligne1[9] = {-1, -1, -1, -1, -1, -1, -1, -1, -1};
        int ligne2[9] = {-1, -1, -1, -1, -1, -1, -1, -1, -1};
        for(int j = 0; j<9; j++){
            int count = 0;
            for(int i = 0; i<9; i++){
                if(notes[i][j][n]){
                    count++ ;
                    if(ligne1[j]==-1){
                        ligne1[j] = i;
                    }
                    else{
                        ligne2[j] = i;
                    }
                }
            }
            if (count == 2){
                i_dep = ligne1[j];
                j_dep = j;
                is_a_chain_column[j] = true;
            }
        }

        bool is_a_chain_zone[9] =
{false,false,false,false,false,false,false,false,false};
        int lignez1[9] = {-1, -1, -1, -1, -1, -1, -1, -1, -1};
        int lignez2[9] = {-1, -1, -1, -1, -1, -1, -1, -1, -1};
        int colonnez1[9] = {-1, -1, -1, -1, -1, -1, -1, -1, -1};
        int colonnez2[9] = {-1, -1, -1, -1, -1, -1, -1, -1, -1};
        for(int z = 0; z<9; z++){
            int count = 0;
            for(int c = 0; c<9; c++){
                if(notes[3*(z/3)+c/3][3*(z%3)+c%3][n]){
                    count++ ;
                    if(colonnez1[c]==-1){
                        lignez1[c] = 3*(z/3)+c/3 ;
                        colonnez1[c] = 3*(z%3)+c%3 ;
                    }
                    else{
                        lignez2[c] = 3*(z/3)+c/3 ;
                        colonnez2[c] = 3*(z%3)+c%3 ;
                    }
                }
            }
            if (count == 2){
                i_dep = lignez1[z];
                j_dep = colonnez1[z];
                is_a_chain_zone[z] = true;
            }
        }

        // Deuxième partie : coloriage
        int i = i_dep ;
        int j = j_dep ;
        int** colour = malloc(9*sizeof(bool*));
        assert(colour!=NULL);
        for(int k = 0; k<9; k++){
            colour[k] = malloc(9*sizeof(bool));
            assert(colour[k]!=NULL);
```

```
            for(int l =0; l<9; l++){
                colour[k][l] = 0 ;
            }
        }
        colour[i_dep][j_dep] = 1 ;


        parcours(i, j, is_a_chain_line, is_a_chain_column, is_a_chain_zone,
        ligne1, ligne2, colonne1,colonne2, lignez1, lignez2, colonnez1,
colonnez2, colour);




    }
    return false;
}



void parcours(int i, int j, bool* is_a_chain_line, bool* is_a_chain_column,
    bool* is_a_chain_zone, int* ligne1, int* ligne2, int* colonne1, int*
colonne2,
    int* lignez1, int* lignez2, int* colonnez1, int* colonnez2, int** colour){

    if(is_a_chain_line[i]){
        int j2 ;
        if(colonne1[i] == j){
            j2 = colonne2[i];
        }
        else{
            j2 = colonne1[i];
        }
        if(colour[i][j2] == colour[i][j]){

        }
    }
}
```

Fichier mon_code/solve.c

```
#include <stdbool.h>
#include <stdio.h>

bool lastRemainingCell(int **grid);
bool lastFreeCell(int **grid);
bool lastPossibleNumber(int **grid);
void printGrid(int** grid);

bool solve(int** grid){
    //printf("Coucou_de_solve\n");
    bool ok = true;
    while (ok) { // tant qu'on trouve un nouvel indice
        ok = lastFreeCell(grid);
        if (!ok) { // si la première méthode échoue, passe à la deuxième
            ok = lastRemainingCell(grid);
            // //printf("coucou1\n");
            if (!ok) { // si la deuxième méthode échoue, passe à la troisième
                // //printf("coucou2\n");
                ok = lastPossibleNumber(grid);

            }
        }
        printGrid(grid);
    }

    // renvoie true si la grille est finie, false sinon
    for(int i = 0; i<9; i++){
        for(int j = 0; j<9; j++){
            //printf("coucou1");
            if(grid[i][j] == 0){
                //printf("\ncoucou2\n");
                return false;
            }
        }
    }
    return true;
}
```

Fichier mon_code/solve_cnf.c

```
#include <assert.h>
#include <bits/types/stack_t.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include<time.h>

typedef struct var_s{
    /* Une variable est de la forme p_i,j,k, elle indique si la case i,j
contient k */
    /* 0 <= i,j < 9 */
    /* 1 <= k <= 9 */
    int i;
    int j;
    int k;
```

```
}var;

typedef struct {
    /* On représente un litéral par une variable (un entier)
    et une positivite (1 si litéral positif, 0 si litéral négatif)
    On représente donc une clause par un tableau de litéraux */
    int nb_lit ;
    var* vars ;
    bool* positif ;
}clause;

struct k_cnf_s {
    int m ; //nb_clauses
    int k ;
    clause* clauses ;
};
typedef struct k_cnf_s* k_cnf;


/* Implémente les ensembles de litéraux */
/* Je pourrais faire des arbres rouge-noir. Je pourrais */
struct lit_set_s {
    var v ;
    bool positif ;
    struct lit_set_s* next ;
};
typedef struct lit_set_s* lit_set ;


// struct maillon_s {
//     k_cnf f ;
//     lit_set* lsptr1 ;
//     lit_set* lsptr2 ;
//     bool val ; //suivant la valeur prise dans la disjonction précédente
//     /* Le suivant dans la file ; vaut NULL pour la queue de file*/
//     struct maillon_s* next ;
//     /* Vaut NULL pour la tête de file */
//     struct maillon_s* prec ;
// } ;
// typedef struct maillon_s* maillon ;

// struct queue_s {
//     maillon head ;
//     maillon tail ;
// };
// typedef struct queue_s* queue ;


void free_clause(clause c);
void print_var(var v);
void print_clause(clause c);
void print_k_cnf(k_cnf f);

void free_k_cnf(k_cnf f){
    for(int a = 0; a<f->m ; a++){
        free_clause(f->clauses[a]);
    }
    free(f->clauses);
    free(f);
}

/* queue queue_create(){
    queue q = malloc(sizeof(struct queue_s));
    assert(q!=NULL);
    q->head = NULL ;
    q->tail = NULL ;
    return q ;
}

void queue_push(queue q, k_cnf f, lit_set* ls_ptr1, lit_set* ls_ptr2, bool b){
    maillon m = malloc(sizeof(struct maillon_s));
    assert(m!=NULL);
    m->f = f ;
    m->lsptr1 = ls_ptr1 ;
    m->lsptr2 = ls_ptr2 ;
    m->val = b ;
    m-> next = NULL ;
    m->prec = q->tail ;
    if(q->tail!=NULL){
        q->tail->next = m ;
    }
    else{
        q->head = m ;
    }
    q->tail = m ;
}

maillon queue_pop(queue q){
    assert(q->head !=NULL);
    maillon m = q->head ;
    q->head = q->head->next ;
    if(q->head!=NULL){
        q->head->prec = NULL ;
    }
```

```
    else{
        q->tail = NULL ;
    }

    return m ;
}

bool queue_is_empty(queue q){
    if(q->head == NULL){
        assert(q->tail==NULL);
        return true ;
    }
    else{
        return false;
    }
}




void free_empty_queue(queue q){
    assert(queue_is_empty(q));
    free(q);
}
*/

bool egal_vars(var v1, var v2){
    return (v1.i == v2.i)&&(v1.j == v2.j) && (v1.k == v2.k);
}


void ls_print(lit_set ls){
    if(ls!=NULL){
        printf("i = %d, j = %d, k = %d, b = %d\n", ls->v.i, ls->v.j, ls->v.k,
ls->positif);
        ls_print(ls->next);
    }
}

bool ls_mem(var v, bool b, lit_set ls){ //ok
    return ((ls!=NULL)&&((egal_vars(v,ls->v) && b == ls->positif)|| ls_mem(v,
b,ls->next)));
}

void ls_union(lit_set ls1, lit_set ls2){
    /* Fait l'union de ls1 et ls2 */
    /* *ls1 est modifié pour contenir cette union */
    /* ls2 n'est pas libéré ni altéré */
    /* ATTENTION : NE FONCTIONNE PAS SI ls1==NULL */
    //printf("Union\n");

    assert(ls1!=NULL);

    if(ls2!=NULL){
        if(ls_mem(ls2->v,ls2->positif,ls1)){
            ls_union(ls1, ls2->next);
        }
        else{
            //printf("ls1 =  ");
            //ls_print(ls1);


            lit_set ls = malloc(sizeof(struct lit_set_s));
            assert(ls!=NULL);
            *ls = *ls1 ;
            ls1->next = ls ;
            ls1->positif = ls2->positif;
            ls1->v = ls2->v ;

            //printf("ls1 =  ");
            //ls_print(ls1);
            ls_union(ls1, ls2->next);
        }
    }
}

lit_set ls_singleton(var v, bool positif){
    lit_set ls = malloc(sizeof(struct lit_set_s));
    assert(ls!=NULL);
    ls->v = v;
    ls->positif = positif ;
    ls->next = NULL ;
    return ls ;
}



void ls_free(lit_set ls){
    if(ls!=NULL){
        ls_free(ls->next);
        free(ls);
    }
}
```

```
bool ls_is_empty(lit_set ls){
    return ls == NULL ;
}

lit_set ls_inter(lit_set ls1, lit_set ls2){
    /* Fait l'intersection de ls1 et ls2 */
    //printf("Inter\n");
    if(ls2!=NULL){
        if(ls_mem(ls2->v,ls2->positif,ls1)){
            lit_set sing = ls_singleton(ls2->v, ls2->positif);
            lit_set inter = ls_inter(ls1,ls2->next);
            if(inter!=NULL){
                ls_union(inter,sing);
                ls_free(sing);
                return inter ;
            }
            else{
                return sing;
            }
        }
        else{
            return ls_inter(ls1,ls2->next);
        }
    }
    else{
        return NULL ;
    }
}


var heuristique_0(k_cnf f){

    /* Choisit une variable uniformément dans la cnf */
    /* Les variables plus présentes ont plus de chances d'être choisies */

    assert(f->m>0);
    int nb_vars_vues = 0;
    var v_choisie ;
    for(int c = 0; c<f->m; c++){
        /* On cherche les clauses à valeurs positives !*/
        for(int i = 0; i<f->clauses[c].nb_lit; i++){
            nb_vars_vues ++ ;
            if (rand()%nb_vars_vues==0){
                v_choisie = f->clauses[c].vars[i];
            }
        }
    }

    return v_choisie;

}



// var heuristique_1(k_cnf f){

//     /* Donne la première variable de la plus petite clause à valeurs
positives */

//     assert(f->m>0);
//     int nb_lit_min = 42;
//     clause c_min ;
//     for(int c = 0; c<f->m; c++){
//         /* On cherche les clauses à valeurs positives !*/
//         if(f->clauses[c].nb_lit<nb_lit_min && f->clauses[c].positif[0]){
//             nb_lit_min = f->clauses[c].nb_lit ;
//             c_min = f->clauses[c] ;
//         }
//     }
//     assert(nb_lit_min>1); // sinon on n'appelerait pas h
//     return c_min.vars[0];
// }

/* On cherche ici la variable qui aurait le plus d'impact */
    /* La variable est présente en positif sur exactement quatre clauses
(ligne, colonne, bloc, case)
    (en effet, si on a résolu une de ces clauses, alors on sait ou placer le
chiffre dans cette zone) */
    /* La présence de la variable négative dépend de la taille de ces zones */
    /* Du fait de l'exploration parallèle des deux cas, l'approche visant à
maximiser le nombre d'occurences
    du litéral négatif est contre-intuitive, il s'agit plutôt de maximiser son
impact à l'aide d'une petite clause */
    /* On va donc simplement affiner l'heuristique précédente */
var heuristique_1(k_cnf f){
    u_int8_t* count = malloc(2*9*9*9*sizeof(u_int8_t));
    assert(count!=NULL);
    for(int i = 0; i<1458; i++){
        count[i]=0;
    }
    for(int c = 0; c<f->m; c++){
        clause cl = f->clauses[c] ;
        for(int i = 0; i<cl.nb_lit; i++){
            int indice = cl.vars[i].i + 9*cl.vars[i].j + 81*cl.vars[i].k ;
```

```
            if(cl.positif[i]){
                indice+=729 ;
            }
            count[indice]++;
        }
    }
    for(int i = 730; i<1458; i++){
        assert(count[i]==0 || count[i]==4);
    }

    assert(f->m>0);
    int nb_lit_min = 42;
    var best_var ;
    for(int c = 0; c<f->m; c++){
        /* On cherche les clauses à valeurs positives !*/
        for(int i = 0; i<f->clauses[c].nb_lit; i++){
            if(f->clauses[c].positif[i]&&(f->clauses[c].nb_lit<nb_lit_min ||
(f->clauses[c].nb_lit==nb_lit_min && count[f->clauses[c].vars[i].i + 9*f-
>clauses[c].vars[i].j + 81* f->clauses[c].vars[i].k] < count[best_var.i +
9*best_var.j + 81*best_var.k]))){
                nb_lit_min = f->clauses[c].nb_lit ;
                best_var = f->clauses[c].vars[i] ;
            }
        }
    }
    assert(nb_lit_min>1); // sinon on n'appelerait pas h
    free(count);
    return best_var;
}


var heuristique_2(k_cnf f){
    u_int8_t* count = malloc(2*9*9*9*sizeof(u_int8_t));
    assert(count!=NULL);
    for(int i = 0; i<1458; i++){
        count[i]=0;
    }
    for(int c = 0; c<f->m; c++){
        clause cl = f->clauses[c] ;
        for(int i = 0; i<cl.nb_lit; i++){
            int indice = cl.vars[i].i + 9*cl.vars[i].j + 81*cl.vars[i].k ;
            if(cl.positif[i]){
                indice+=729 ;
            }
            count[indice]++;
        }
    }
    for(int i = 730; i<1458; i++){
        assert(count[i]==0 || count[i]==4);
    }

    assert(f->m>0);
    int nb_lit_max = 42;
    var best_var ;
    for(int c = 0; c<f->m; c++){
        /* On cherche les clauses à valeurs positives !*/
        for(int i = 0; i<f->clauses[c].nb_lit; i++){
            if(f->clauses[c].positif[i]&&(f->clauses[c].nb_lit<nb_lit_max ||
(f->clauses[c].nb_lit==nb_lit_max && count[f->clauses[c].vars[i].i + 9*f-
>clauses[c].vars[i].j + 81* f->clauses[c].vars[i].k] > count[best_var.i +
9*best_var.j + 81*best_var.k]))){
                nb_lit_max = f->clauses[c].nb_lit ;
                best_var = f->clauses[c].vars[i] ;
            }
        }
    }
    assert(nb_lit_max<=9); // sinon on n'appelerait pas h
    free(count);
    return best_var;
}

void substitue(var v, bool b, k_cnf f){
    /* Met la variable v à la valeur b dans f */
    /* Si c est une clause de f et v la i-ème variable de c, */
    /* Alors c est supprimée si c.positif[i] == b */
    /* Et v est retirée sinon */


    //printf("Subitute :");
    print_var(v) ;
    //printf(", b= %d\n",b);


    for(int c = 0; c<f->m; c++){
        for(int i = 0; i<f->clauses[c].nb_lit; i++){
            if(egal_vars(v,f->clauses[c].vars[i])){
                if(f->clauses[c].positif[i]==b){
                    // for(int i = 0; i<f->clauses[c].nb_lit; i++){
                    //     ls_union(ls, ls_singleton(f->clauses[c].vars[i]));
                    //     /* Merde il faut la positivité aussi */
                    // }
                    //printf("On supprime une clause\n");
                    free_clause(f->clauses[c]);
                    if(c<f->m-1){
```

```
                    f->clauses[c] = f->clauses[f->m-1];
                }
                f->m -- ;
                c-- ;
                break ;
            }
            else{
                //printf("On supprime une occurence d'une variable\n");
                if(i<f->clauses[c].nb_lit-1){
                    f->clauses[c].vars[i] = f->clauses[c].vars[f-
>clauses[c].nb_lit-1];
                    f->clauses[c].positif[i] = f->clauses[c].positif[f-
>clauses[c].nb_lit-1];
                }
                f->clauses[c].nb_lit -- ;
                i-- ;

            }
        }
    }
}

lit_set* quine(k_cnf f){ //ok
    /* Effectue une itération de Quine sur f */
    /* Modifie f au passage */
    /* Renvoie un singleton contenant la variable si une variable est trouvée
*/
    /* Renvoie un pointeur sur l'ensemble vide si aucune variable est trouvée
*/
    /* Renvoie NULL si la formule est insatisfiable car une clause est vide */

    //printf("########### Quine : ############\n");
    for(int c = 0; c<f->m; c++){
        //printf("%d   ", f->clauses[c].nb_lit);
        print_clause(f->clauses[c]);
        if(f->clauses[c].nb_lit == 1){
            lit_set* found = malloc(sizeof(lit_set));
            assert(found!=NULL);
            //printf("Clause à un littéral trouvée : i = %d, j = %d, k = %d, b
= %d\n", f->clauses[c].vars[0].i,f->clauses[c].vars[0].j, f-
>clauses[c].vars[0].k, f->clauses[c].positif[0]);
            *found = ls_singleton(f->clauses[c].vars[0], f-
>clauses[c].positif[0]);
            substitue(f->clauses[c].vars[0], f->clauses[c].positif[0],f);
            return found ;
        }
        else if(f->clauses[c].nb_lit == 0){
            //printf("Formule insatisfiable\n");

            return NULL ;
        }
    }
    lit_set* found = malloc(sizeof(lit_set));
    assert(found!=NULL);
    *found = NULL ;
    return found ;
}

bool*** copy_notes(bool*** notes){
    bool*** t = malloc(9*sizeof(bool**));
    assert(t!=NULL);
    for(int i = 0; i<9; i++){
        t[i] = malloc(9*sizeof(bool*));
        assert(t[i]!=NULL);
        for(int j = 0; j<9; j++){
            t[i][j] = malloc(9*sizeof(bool));
            assert(t[i][j]!=NULL);
            for(int k = 0; k<9; k++){
                t[i][j][k] = notes[i][j][k];
            }
        }
    }
    return t;
}

k_cnf k_cnf_copy(k_cnf f){
    //printf("Begin k_cnf_copy\n");
    k_cnf g = malloc(sizeof(struct k_cnf_s));
    assert(g!=NULL);
    g->m = f->m ;
    g->k = f->k ;
    g->clauses = malloc(g->m * sizeof(clause));
    assert(g->clauses!=NULL);
    for(int c = 0; c<f->m; c++){
        clause c2 ;
        c2.nb_lit = f->clauses[c].nb_lit ;

        c2.vars = malloc(c2.nb_lit*sizeof(var));
        assert(c2.vars!=NULL);
        c2.positif = malloc(c2.nb_lit*sizeof(bool));
        assert(c2.positif!=NULL);

        for(int i = 0; i<c2.nb_lit;i++){
```

```
            c2.vars[i] = f->clauses[c].vars[i];
            c2.positif[i] = f->clauses[c].positif[i];
        }

        g->clauses[c] = c2 ;

    }
    return g;
}



// void solve_cnf_aux(k_cnf f, lit_set* ls_ptr1, lit_set* ls_ptr2, queue q,
var(*h)(k_cnf), int* nb_disjonctions, int* nb_quines, int profondeur){
//     /* Effectue l'opération élémentaire associée à la formule f  */
//     /* Les variables trouvées sont mises dans modified_t */
//     /* Si modified_t inter modified_f est non vide, on s 'arrête */
//     /* Le booléen finished_quine sert à gérer les déséquilibres dans le
nombre de quines possibles */


//     //printf("Solve_cnf_aux\n");

//     /* On ne continue pas si on a trouvé quelque chose */


// }


lit_set disjonction(k_cnf f, var(*h)(k_cnf), int* nb_disjonctions, int*
nb_quines, int profondeur){
    /* Résout la formule de logique propositionelle f associée à la grille grid
*/
    /* Avec l'algortihme de Quine */
    /* L'heuristique utilisée est passée en argument*/
    /* Renvoie l'ensemble des litéraux communes (qui sont donc nécéssairement
vrais)*/

    //printf("Disjonction !\n");
    if(profondeur<3){
        printf("profondeur = %d,  m = %d\n",profondeur, f->m);
    }
    nb_disjonctions[profondeur]++;



    //printf("File créée\n");
```

```
    /* Heuristique passée en argument, trouve la variable pour la disjonction
*/
    var v = h(f);
    //printf("Variable trouvée\n");

    k_cnf f_true = k_cnf_copy(f);
    k_cnf f_false = k_cnf_copy(f);

    lit_set modified_t = ls_singleton(v, true);
    substitue(v, true, f_true);

    lit_set modified_f =  ls_singleton(v,false);
    substitue(v, false, f_false);



    bool finished_quine_t = false ;

    bool finished_quine_f = false ;
    //printf("Coucou\n");

    lit_set inter = ls_inter(modified_t, modified_f);
    while( ls_is_empty(inter)){

        if(!finished_quine_t){
            if(f_true->m == 0){
                for(lit_set c = modified_t; c !=NULL; c = c->next){
                    substitue(c->v, c->positif, f);
                }
                ls_free(modified_f);
                free_k_cnf(f_false);
                free_k_cnf(f_true);
                return modified_t ;
            }

            lit_set* found = quine(f_true);
            nb_quines[profondeur+1]++;


            /* Si la clause est satisfiable, on continue*/
            if(found!=NULL){
                if(*found == NULL){
                    free(found) ;
                    finished_quine_t = true ;
                }
                else{
                    //printf("Coucou\n");
```

```
                // Par construction, modified_t!=NULL
                ls_union(modified_t, *found);
                //printf("Coucou\n");
                ls_free(*found);
                free(found);
            }

        }
        else{
            /* Attention, ce qu'on va faire n'a aucun sens, mais permet de
"valider" l'autre clause (puisque celle-là est fausse)*/
            /* N.B. On ne dit pas que l'autre clause est forcément
satisfiable, juste que si la clause mère est satisfiable, alors l'autre l'est
aussi*/
            //printf("La formule est insatisfiable d'après Quine\n");
            assert(modified_f!=NULL);
            //printf("modified_t = \n");
            //ls_print(modified_t);
            //printf("modified_f = \n");
            //ls_print(modified_f);
            assert(modified_t!=NULL);
            ls_union(modified_t, modified_f);
            //printf("modified_t = \n");
            //ls_print(modified_t);
            //printf("modified_f = \n");
            //ls_print(modified_f);
            lit_set inter = ls_inter(modified_f,modified_t);
            assert(inter!=NULL);
            ls_free(inter);
        }
    }
    else if(!finished_quine_f){
        if(f_false->m == 0){
            for(lit_set c = modified_f; c !=NULL; c = c->next){
                substitue(c->v, c->positif, f);
            }
            ls_free(modified_t);
            free_k_cnf(f_false);
            free_k_cnf(f_true);
            return modified_f ;
        }

        lit_set* found = quine(f_false);
        nb_quines[profondeur+1]++;


        /* Si la clause est satisfiable, on continue*/
        if(found!=NULL){
            if(*found == NULL){
                free(found) ;
                finished_quine_f = true ;
            }
            else{
                //printf("Coucou3\n");
                ls_union(modified_f, *found);

                ls_free(*found);
                free(found);
            }

        }
        else{
            /* Attention, ce qu'on va faire n'a aucun sens, mais permet de
"valider" l'autre clause (puisque celle-là est fausse)*/
            /* N.B. On ne dit pas que l'autre clause est forcément
satisfiable, juste que si la clause mère est satisfiable, alors l'autre l'est
aussi*/
            //printf("La formule est insatisfiable d'après Quine\n");
            //printf("Coucou4\n");
            ls_union(modified_f, modified_t);


        }


    }

    else{

        if(rand()%2 == 0){

            lit_set ls = disjonction(f_true,h, nb_disjonctions, nb_quines,
profondeur+1);
            //ls_print(ls);
            if(!ls_is_empty(ls)){
                //printf("ls est non vide\n");
                //printf("Coucou5\n");
                ls_union(modified_t, ls);
                ls_free(ls);

            }
            else{
                /* Alors f est nécéssairement insatisfiable*/
                for(lit_set c = modified_f; c !=NULL; c = c->next){
                    substitue(c->v, c->positif, f);
```

```
                    }
                    ls_free(ls);
                    ls_free(modified_t);
                    free_k_cnf(f_false);
                    free_k_cnf(f_true);
                    return modified_f ;
                }

                finished_quine_t = false;
            }
            else{
                lit_set ls = disjonction(f_false,h, nb_disjonctions, nb_quines,
profondeur+1);
                //ls_print(ls);
                if(!ls_is_empty(ls)){
                    //printf("ls est non vide\n");
                    //printf("Coucou6\n");
                    ls_union(modified_f, ls);
                    ls_free(ls);

                }
                else{
                    /* Alors f est nécéssairement insatisfiable*/
                    for(lit_set c = modified_t; c !=NULL; c = c->next){
                        substitue(c->v, c->positif, f);
                    }
                    ls_free(ls);
                    ls_free(modified_f);
                    free_k_cnf(f_false);
                    free_k_cnf(f_true);
                    return modified_t ;
                }

                finished_quine_f = false;
            }
        }

        lit_set temp = inter ;
        inter = ls_inter(modified_t, modified_f);
        ls_free(temp);
    }


    //printf("ls = \n");
    //ls_print(ls);
    assert(inter != NULL);
    for(lit_set c = inter; c !=NULL; c = c->next){
        substitue(c->v, c->positif, f);
    }


    ls_free(modified_f);
    ls_free(modified_t);
    free_k_cnf(f_false);
    free_k_cnf(f_true);

    return inter ;

}



void solve_cnf(k_cnf f, var(*h)(k_cnf), int* nb_disjonctions, int* nb_quines){
    /* Fonctionne sans disjonction ! */

    /* nb_disjonctions contient le nombre de disjonction à chaque profondeur */
    /* nb_quines contient le nombre de quines à chaque profondeur*/



    while(f->m>0){
        //printf(" m = %d\n", f->m);

        lit_set* found = quine(f) ;

        nb_quines[0]++ ;
        print_k_cnf(f);
        if(found==NULL){

            //printf("La formule initiale est insatisfiable !\n");
            fflush(stdout);
            assert(42==69);
        }
        else{
            if(ls_is_empty(*found)){

                lit_set ls = disjonction(f, h, nb_disjonctions, nb_quines, 0);

                //printf (" ls = \n");
                //ls_print(ls);
                ls_free(ls);
                print_k_cnf(f);
            }
            else{
```

```
                ls_free(*found);
            }
            free(found);

        }
    }

}


/* Questions : */
// - Faut-il compter/différencier les quines vrais et faux ?
// (les quines faux sont des applications de règles)
```

Fichier mon_code/solve_notes.c

```
#include <assert.h>
#include <stdbool.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

bool lastFreeCell(grid_t g);
int hiddenSingle(grid_t g);
bool nakedSingle(grid_t g);

bool nakedPair(grid_t g);
bool nakedTriple(grid_t g);
bool hiddenPair(grid_t g);
bool hiddenTriple(grid_t g);
bool pointingPair(grid_t g);
bool boxLineReduction(grid_t g);
bool x_wing(grid_t g);
bool y_wing(grid_t g);
bool swordfish(grid_t g);

void printGrid(int **grid);
void print_tab_float(float *tab, int size);

/* Entrées : une grille, ses notes
Sortie : true si la grille a été résolue, false sinon
Effet de bord : la grille contient autant d'indices que possible,
les notes sont le plus vide possible
Remarque : la séparation de la résolution d'une grille avec le main permet à la
fois de simplifier la lecture du code, et d'utiliser le backtracking pour finir
la grille */
bool est_ok(int** grid){
  for(int i = 0; i<9; i++){
    int num[9] = {0,0,0,0,0,0,0,0,0};
    for(int j = 0; j<9; j++){
      if(grid[i][j] != 0){
        num[grid[i][j]-1] ++ ;
      }
    }
    for(int j = 0; j<9; j++){
      if(num[j]>1){
        return false ;
      }
    }
  }
  for(int j = 0; j<9; j++){
    int num[9] = {0,0,0,0,0,0,0,0,0};
    for(int i = 0; i<9; i++){
      if(grid[i][j] != 0){
        num[grid[i][j]-1] ++ ;
      }
    }
    for(int i = 0; i<9; i++){
      if(num[i]>1){
        return false ;
      }
    }
  }
  for(int z = 0; z<9; z++){
    int num[9] = {0,0,0,0,0,0,0,0,0};
    for(int c = 0; c<9; c++){
      if(grid[3*(z/3) + c/3][3*(z%3) + c%3] != 0){
        num[grid[3*(z/3) + c/3][3*(z%3) + c%3]-1] ++ ;
      }
    }
    for(int j = 0; j<9; j++){
      if(num[j]>1){
        return false ;
      }
    }
  }
  return true;
}


bool solve_notes(grid_t g, bool(**techniques)(grid_t g), int p) {
  /* Entrées : - g la grille à traiter */
```

```
  /* - techniques un tableau de pointeurs de fonctions, */
  /* qui sont les techniques de résolution (hors backtracking) */
  /* - p le nombre de techniques */
  /* Applique les techniques par ordre croissant de difficulté */

  int i = 0 ;
  while(i<p){
    if(techniques[i](g)){
      g->nb_techniques[i] ++ ;
      i = 0;
    }
    else{
      i++;
    }
    if(!est_ok(g->grid)){
      return false;
    }
  }

  // teste successivement les techniques, dans l'ordre croissant des
difficultés
  // bool ok = true ;
  // while (ok) {
  //   if(!est_ok(g->grid)){
  //     return false;
  //   }
  //   // easy
  //   //printf("SolveNotes\n");
  //   ok = lastFreeCell(g);
  //   //ok = false ;
  //   if(!ok){
  //     int nb_pos = hiddenSingle(g);
  //     ok = (nb_pos >= 1);
  //     if (!ok) {
  //       ok = nakedSingle(g);
  //       if (!ok) {
  //         // medium

  //         ok = nakedPair(g);
  //         if (!ok) {
  //           //printf("coucou4\n");
  //           // fflush(stdout);
  //           ok = nakedTriple(g);

  //           if (!ok) {
  //             //printf("coucou5\n");
  //             // fflush(stdout);
  //             ok = pointingPair(g);

  //             if (!ok) {
  //               //printf("coucou6\n");
  //               // fflush(stdout);

  //               ok = boxLineReduction(g);

  //               if (!ok) {
  //                 //printf("coucou7\n");
  //                 // fflush(stdout);
  //                 ok = hiddenPair(g);
  //                 if (!ok) {
  //                   //printf("coucou8\n");
  //                   // fflush(stdout);
  //                   ok = hiddenTriple(g);
  //                   if (!ok) {
  //                     //printf("coucou9\n");
  //                     // fflush(stdout);
  //                     ok = x_wing(g);
  //                     // //printf("coucou10\n");
  //                     // fflush(stdout);
  //                     if(!ok){
  //                       ok = y_wing(g);
  //                       if(!ok){
  //                         ok = swordfish(g);
  //                         if(ok){
  //                           g->nb_techniques[11] ++ ;
  //                         }
  //                       }
  //                       else{
  //                         g->nb_techniques[10] ++ ;
  //                       }
  //                     }
  //                     else{
  //                       g->nb_techniques[9]++ ;
  //                     }
  //                   }
  //                   else {
  //                     g->nb_techniques[8]++;
  //                   }
  //                 } else {
  //                   g->nb_techniques[7]++;
  //                 }
  //               } else {
  //                 g->nb_techniques[6]++;
```

```
  //               }
  //             } else {
  //               g->nb_techniques[5]++;
  //             }
  //           } else {
  //             g->nb_techniques[4]++;
  //           }
  //         } else {
  //           g->nb_techniques[3]++;
  //         }
  //       } else {
  //         g->nb_techniques[1]++;
  //       }
  //     } else {
  //       /* Le décalage d'indices vient d'une revue de l'aordre de difficulté
des techniques */
  //       g->nb_techniques[2]+= nb_pos/9.;
  //     }
  //   }
  //   else{
  //     g->nb_techniques[0]++;


  //   }
  // }
  // print_tab(nb_techniques,10);
  //  renvoie true si la grille est finie, false sinon
  // //printf("coucou42\n");
  // fflush(stdout);

  bool finished  = true ;
  for(int c = 0; c<81; c++){
    if(g->grid[c/9][c%9] == 0){
      finished = false;
      break ;
    }
  }

  if(!finished){
    //printf("SolveNotes pas fini\n");
    //printGrid(g->grid);
  }

  return finished ;

}
```

Fichier mon_code/solve_simple_notes_backtrack.c

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

void printGrid(int **grid); // ok

bool ***createNotes();
void updateNotes(grid_t g, int row, int col);

void print_tab_float(float *tab, int size);

bool solve(int **grid);
bool solve_notes(grid_t g, bool(**techniques)(grid_t g), int n);
bool backtrack(grid_t g, bool(**techniques)(grid_t g), int n);

void initialize_notes(grid_t g);

void print_notes(bool ***notes);
void free_notes(bool*** notes);

float consequences_new_number(int** grid, bool*** notes,int i, int j, float*
nb_techniques);

void solve_simple_notes_backtrack(grid_t g, bool(**techniques)(grid_t g), int
n){

    bool finished = false ;

    //finished = solve(grid);
    if(!finished){
        //printf(" on utilise les notes\n");
        initialize_notes(g);
        //printGrid(g->grid);
        assert(g->notes!=NULL);
        //print_notes(g->notes);
        finished = solve_notes(g, techniques, n);
```

```
        //print_tab_float(g->nb_techniques, 13);
        // si les techniques ne suffisent pas,
        // on passe au backtracking
        if (!finished){
            //printGrid(g->grid);
            finished = backtrack(g, techniques, n);
            if (!finished){
                //printGrid(g->grid);
                print_tab_float(g->nb_techniques,13);
            }
            assert(finished);
        }
        free_notes(g->notes);
    }
}
```

Fichier mon_code/sudoku_to_cnf.c

```
#include<stdlib.h>
#include<assert.h>
#include<stdbool.h>
#include<stdio.h>

typedef struct var_s{
    /* Une variable est de la forme p_i,j,k, elle indique si la case i,j
contient k */
    /* 0 <= i,j < 9 */
    /* 1 <= k <= 9 */
    int i;
    int j;
    int k;
}var;

typedef struct {
    /* On représente un litéral par une variable (un entier)
    et une positivite (1 si litéral positif, 0 si litéral négatif)
    On représente donc une clause par un tableau de litéraux */
    int nb_lit ;
    var* vars ;
    bool* positif ;
}clause;


typedef struct {
    /* Ce type est beaucoup moins général ; le filtre est spécifique à cet
usage */
    int nb_var ;
    var* vars ;
    bool* filtre ;
}clause_1in9;


struct k_cnf_s {
    int m ; //nb_clauses
    int k ;
    clause* clauses ;
};
typedef struct k_cnf_s* k_cnf;


void free_clause(clause c){
    free(c.vars);
    free(c.positif);
}

void free_clause_1in9(clause_1in9 c){
    free(c.vars);
    free(c.filtre);
}

void print_var(var v){
    //printf("i = %d, j = %d, k = %d\n", v.i, v.j, v.k);
}

void print_clause(clause c){
    for(int b = 0; b<c.nb_lit;b++){
            if(c.positif[b]){
                //printf("x_%d,%d,%d ", c.vars[b].i, c.vars[b].j, c.vars[b].k);
            }
            else{
                //printf("not x_%d,%d,%d ", c.vars[b].i, c.vars[b].j,
c.vars[b].k);
            }
            if(b<c.nb_lit-1){
                //printf("or ");
            }
        }
    //printf("\n");
}

void print_k_cnf(k_cnf f){
    //printf("m = %d\n",f->m);
    //printf("k = %d\n", f->k);
    for(int a = 0; a<f->m; a++){
        print_clause(f->clauses[a]);
    }
```

```
}


k_cnf sudoku_to_cnf(int** grid){
    /* génère une cnf à partir d'une grille de sudoku */
    /* La tactique qui consiste à encoder les règles puis les indices ne
fonctionne pas
    car un sudoku plus compliqué aurait donc moins de contraintes pour le même
nombre d'indices...
    et de toute façon cela riviendrait juste à compter les indices initiaux */

    clause_1in9* clauses_1in9 = malloc(324*sizeof(clause_1in9));
    assert(clauses_1in9!=NULL);

    /* La transformation se fera donc en trois étapes :
    1 - écriture des clauses 1-dans-9 imposées par les règles (324 au total =
81 pour les lignes, colonnes, zones et cases)
    2 - élagage de ces clauses : chaque indice donné nous permet
        - de supprimer 4 clauses (celles ou la variable mise à vrai apparait)
        - de supprimer toutes les autres variables de la ligne, colonne, zone
et tour de l'indice
        il y a au plus 28 variables à retirer des clauses restantes
    on ne considère pas que les clauses réduites à une variable forment de
nouveaux indices,
    car cela correspondrait à dérouler les naked single et hidden single */
    /* Remarque : la fusion de ces deux étapes permettrait de gagner en
efficacité, mais perdrait en clarté */

    /* Étape 1 :*/
    /* Cases */
    for(int i = 0; i<9; i++){
        for(int j = 0; j<9; j++){
            var* vars = malloc(9*sizeof(var));
            assert(vars!=NULL);
            bool* filtre = malloc(9*sizeof(bool));
            assert(filtre!=NULL);
            for(int k = 0; k<9; k++){
                filtre[k] = true;
                var v = {.i = i, .j=j, .k=k};
                vars[k] = v;
                print_var(vars[k]);
            }
            clause_1in9 c = {.nb_var = 9, .vars = vars, .filtre = filtre};
            clauses_1in9[9*i+j] = c;
        }
    }

    /* Lignes */
    for(int i = 0; i<9; i++){
        for(int k = 0; k<9; k++){
            var* vars = malloc(9*sizeof(var));
            assert(vars!=NULL);
            bool* filtre = malloc(9*sizeof(bool));
            assert(filtre!=NULL);
            for(int j = 0; j<9; j++){
                filtre[j] = true;
                var v = {.i = i, .j=j, .k=k};
                vars[j] = v;
                print_var(vars[j]);
            }
            clause_1in9 c = {.nb_var = 9, .vars = vars, .filtre = filtre};
            clauses_1in9[81 + 9*i+k] = c;
        }
    }

    /* Colonnes */
    for(int j = 0; j<9; j++){
        for(int k = 0; k<9; k++){
            var* vars = malloc(9*sizeof(var));
            assert(vars!=NULL);
            bool* filtre = malloc(9*sizeof(bool));
            assert(filtre!=NULL);
            for(int i = 0; i<9; i++){
                filtre[i] = true;
                var v = {.i = i, .j=j, .k=k};
                vars[i] = v;
                print_var(vars[i]);
            }
            clause_1in9 c = {.nb_var = 9, .vars = vars, .filtre = filtre};
            clauses_1in9[162 + 9*j+k] = c;
        }
    }

    /* Zones */
    for(int z = 0; z<9; z++){
        for(int k = 0; k<9; k++){
            var* vars = malloc(9*sizeof(var));
            assert(vars!=NULL);
            bool* filtre = malloc(9*sizeof(bool));
            assert(filtre!=NULL);
            for(int c = 0; c<9; c++){
                filtre[c] = true;
                var v = {.i = 3*(z/3)+c/3, .j=3*(z%3)+c%3, .k=k};
                vars[c] = v;
```

```
                print_var(vars[c]);
            }
            clause_1in9 cl = {.nb_var = 9, .vars = vars, .filtre = filtre};
            clauses_1in9[243 + 9*z+k] = cl;
        }
    }



    /* Étape 2 : */
    bool* filtre = malloc(324*sizeof(bool));
    assert(filtre!=NULL);
    for(int i = 0; i<324; i++){
        filtre[i] = true ;
    }
    /* On ne peut pas retirer les éléments du tableau à proprement parler, donc
on filtre */


    for(int i = 0; i<9; i++){
        for(int j = 0; j<9; j++){
            if(grid[i][j]>0){
                int k = grid[i][j] -1 ;
                int z = 3*(i/3) + j/3 ;
                int c = 3*(i%3) + j%3 ;
                filtre[9*i+j] = false ;
                filtre[81 + 9*i+k] = false ;
                filtre[162 + 9*j+k] = false ;
                filtre[243 + 9*z+k] = false ;

                // on retire ces variables
                for(int k2 = 0; k2<9; k2++){
                    clauses_1in9[81 + 9*i+k2].filtre[j] = false ;
                    clauses_1in9[162 + 9*j+k2].filtre[i] = false ;
                    clauses_1in9[243 + 9*z+k2].filtre[c] = false ;

                }
                for(int i2 = 0; i2<9; i2++){
                    clauses_1in9[9*i2+j].filtre[k] = false ;
                    clauses_1in9[81 + 9*i2+k].filtre[j] = false ;
                    int z2 = 3*(i2/3)+j/3 ;
                    int c2 = 3*(i2%3)+(j%3) ;
                    assert(z2<9);
                    assert(c2<9) ;
                    clauses_1in9[243 + 9*z2+k].filtre[c2] = false ;
                }
                for(int j2 = 0; j2<9; j2++){
                    clauses_1in9[9*i+j2].filtre[k] = false ;
                    clauses_1in9[162 + 9*j2+k].filtre[i] = false ;
                    clauses_1in9[243 + 9*(3*(i/3)+j2/3)+k].filtre[3*(i%3)+j2%3]
= false ;
                }
                for(int c2 = 0; c2<9; c2++){
                    int i2 = 3*(z/3)+c2/3 ;
                    int j2 = 3*(z%3)+c2%3 ;
                    clauses_1in9[9*i2+j2].filtre[k] = false ;
                    clauses_1in9[81 + 9*i2+k].filtre[j2] = false ;
                    clauses_1in9[162 + 9*j2+k].filtre[i2] = false ;

                }
            }
        }
    }


    /* On crée la k_cnf */

    int m = 0; //nb_clauses

    // tableau pouvant stocker le nombre maximal de clauses(sudoku vide)
    // il sera redimensionné
    clause* clauses = malloc(324*37*sizeof(clause));
    assert(clauses!=NULL);



    for(int a = 0; a<324; a++){
        if(filtre[a]){
            int nb_lit = 0;
            for(int b = 0; b<9; b++){
                if(clauses_1in9[a].filtre[b]){
                    nb_lit++ ;
                }
            }
            // si il y a au moins une variable
            if(nb_lit >0){
                var* vars = malloc(nb_lit*sizeof(var));
                assert(vars!=NULL);
                bool* positif = malloc(nb_lit*sizeof(bool));
                assert(positif!=NULL);
                int i = 0;
                for(int b = 0; b<9; b++){
                    if(clauses_1in9[a].filtre[b]){
                        vars[i] = clauses_1in9[a].vars[b] ;
```

```
                    print_var(vars[i]);
                    positif[i] = true ;
                    i++ ;
                }
            }
            clause c = {.vars = vars, .nb_lit = nb_lit, .positif =
positif} ;
            clauses[m] = c;
            m++;

            for(int i = 0; i<nb_lit; i++){
                for(int j = 0; j<nb_lit; j++){
                    if(i<j){
                        var* vars_2 = malloc(2*sizeof(var));
                        assert(vars_2!=NULL);
                        bool* pos_2 = malloc(2*sizeof(bool));
                        assert(pos_2!=NULL);
                        vars_2[0] = vars[i];
                        vars_2[1] = vars[j];
                        pos_2[0] = false ;
                        pos_2[1] = false ;
                        clause c_2 =
{.nb_lit=2, .positif=pos_2, .vars=vars_2} ;
                        clauses[m] = c_2;
                        m++ ;
                    }
                }
            }
        }
    }
    //printf("m = %d\n",m);

    clause* clauses_redimensionne = malloc(m*sizeof(clause));
    assert(clauses_redimensionne!=NULL);
    for(int i = 0; i<m; i++){
        clauses_redimensionne[i]=clauses[i];
    }

    /* Mettre des free ! */
    // for(int i = 0; i<m; i++){
    //     free_clause(clauses[i]);
    // }
    free(clauses);
    for(int i =0; i<324; i++){
        free_clause_1in9(clauses_1in9[i]);
    }
    free(clauses_1in9);
    free(filtre);

    k_cnf f = malloc(sizeof(struct k_cnf_s));
    assert(f!=NULL);
    f->m = m;
    f->k = 9;
    f->clauses = clauses_redimensionne ;

    print_k_cnf(f);
    return f;
}
```

Fichier mon_code/swordfish.c

```
#include <stdbool.h>
#include<stdio.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

bool swordfish(grid_t g){
    bool trouve = false ;

    // pour chaque chiffre
    for(int n = 1; n <= 9; n++){
        //printf("%d\n",n);
        // compteur des occurences de n dans les notes dans chaque ligne
        int countl[9] = {0,0,0,0,0,0,0,0,0};


        // on parcourt les lignes
        for(int i = 0; i<9; i++){
            // on parcourt la ligne choisie
            for(int j = 0; j<9; j++){
                // si la case ij peut accueillir n (Attention au décalage
d'indices !)
                if (g->notes[i][j][n-1]){
                    countl[i] ++ ;
                }
            }
        }
        // pour toutes les paires de lignes
        for(int i1 = 0; i1<9; i1++){
            if(countl[i1] == 2 || countl[i1]==3){
```

```
                for(int i2 = i1+1; i2<9; i2++){
                    if(countl[i2] == 2 || countl[i2]==3){
                        for(int i3 = i2+1; i3<9; i3++){
                            if(countl[i3] == 2 || countl[i3]==3){
                                //printf("Triplet %d %d %d\n", i1, i2, i3);
                                // si elles remplissent les conditions :
                                // - différentes
                                // - avec deux indices chacune
                                // - avec leurs deux indices sur la mêm colonne
                                // alors on applique la technique
                                bool place[9] =
{false,false,false,false,false,false,false,false,false};
                                int count = 0;
                                for(int j = 0; j<9; j++){
                                    if (g->notes[i1][j][n-1]||g->notes[i2][j]
[n-1]||g->notes[i3][j][n-1]){
                                        place[j] = true;
                                        count ++;
                                    }
                                }
                                if(count <= 3){
                                    // normalement count == 3 mais dans le
doute
                                    for(int j = 0; j<9; j++){
                                        // si on ne modifie rien, il ne faut
pas renvoyer true !
                                        if(place[j]){
                                            for(int i = 0; i<9; i++){
                                                if((i != i1) && (i != i2) &&
(i!=i3) && g->notes[i][j][n-1]){
                                                    g->notes[i][j][n-1] =
false ;
                                                    //printf("Technique :
swordfish\n");
                                                    //printf("On retire %d en
%d %d\n", n, i, j);
                                                    trouve = true ;
                                                }
                                            }
                                        }
                                    }
                                    if(trouve){
                                        return true;
                                    }
                                }
                            }
                        }
                    }
                }
            }

            // compteur des occurences de n dans les notes dans chaque colonne
            int countc[9] = {0,0,0,0,0,0,0,0,0};


            // on parcourt les colonnes
            for(int j = 0; j<9; j++){

                // on parcourt la colonne choisie
                for(int i = 0; i<9; i++){
                    // si la case ij peut accueillir n (Attention au décalage
d'indices !)
                    if (g->notes[i][j][n-1]){
                        countc[j] ++ ;
                    }
                }
            }


            // pour toutes les paires de colonnes
            for(int j1 = 0; j1<9; j1++){
                if(countc[j1] == 2 || countc[j1]==3){
                    for(int j2 = j1+1; j2<9; j2++){
                        if(countc[j2] == 2 || countc[j2]==3){
                            for(int j3 = j2+1; j3<9; j3++){
                                if(countc[j3] == 2 || countc[j3]==3){
                                    //printf("Triplet %d %d %d\n", j1, j2, j3);
                                    // si elles remplissent les conditions :
                                    // - différentes
                                    // - avec deux indices chacune
                                    // - avec leurs deux indices sur la même ligne
                                    // alors on applique la technique

                                    bool place[9] =
{false,false,false,false,false,false,false,false,false};
                                    int count = 0;
                                    for(int i = 0; i<9; i++){
                                        if (g->notes[i][j1][n-1]||g->notes[i][j2]
[n-1]||g->notes[i][j3][n-1]){
                                            place[i] = true;
                                            count ++;
                                        }
```

```
                                }


                                if(count<=3){
                                    for(int i = 0; i <9; i++){
                                        if(place[i]){
                                            for(int j = 0; j<9; j++){
                                                // si on ne modifie rien, il ne
faut pas renvoyer true !
                                                if((j != j1) && (j != j2) &&
(j!=j3) && g->notes[i][j][n-1]){
                                                    g->notes[i][j][n-1] =
false ;
                                                    //printf("Technique :
swordfish\n");
                                                    //printf("On retire %d en
%d %d\n", n, i, j);
                                                    trouve = true ;
                                                }
                                            }
                                        }
                                    }
                                    if(trouve){
                                        return true ;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    return trouve ;
}
```

Fichier mon_code/updateNotes.c

```
#include <assert.h>
#include <stdbool.h>



struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

// l'appel de cette fonction pour toute les cases permet un premier remplissage
de la grille
void updateNotes(grid_t g, int row, int col){
    // Entrées : une grille, deux indices row et col
    // Sorties : aucune ; les cellules du tableau de notes sont modifées sur la
ligne, la colonne et la zone de row-col pour retirer grid[row][col] des notes
    int value = g->grid[row][col] ;
    if(value!=0){
        for(int i = 0; i<9; i++){
            g->notes[row][col][i] = false ;
        }



        int iPos = value - 1 ; // indice dans notes
        for(int i = 0; i < 9; i ++){
            g->notes[i][col][iPos] = false;
        }
        for(int j = 0; j < 9; j ++){
            g->notes[row][j][iPos] = false;
        }
        for(int i = 0; i < 3; i ++){
            for(int j =0; j<3; j++){
                g->notes[3*(row/3) + i][3*(col/3) + j][iPos] = false;
            }
        }
    }
}
```

Fichier mon_code/x_wing.c

```
#include <stdbool.h>
#include<stdio.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

bool x_wing(grid_t g){
    bool trouve = false ;


    // pour chaque chiffre
    for(int n = 1; n <= 9; n++){
```

```
        // compteur des occurences de n dans les notes dans chaque ligne
        int countl[9] = {0,0,0,0,0,0,0,0,0};

        // indices de deux colonnes contenat des indices (utile si count[i] =
2)
        int colonne1[9] = {-1, -1, -1, -1, -1, -1, -1, -1, -1};
        int colonne2[9] = {-1, -1, -1, -1, -1, -1, -1, -1, -1};

        // on parcourt les lignes
        for(int i = 0; i<9; i++){
            // on parcourt la ligne choisie
            for(int j = 0; j<9; j++){
                // si la case ij peut accueillir n (Attention au décalage
d'indices !)
                if (g->notes[i][j][n-1]){
                    countl[i] ++ ;
                    if(colonne1[i]==(-1)){
                        colonne1[i] = j;
                    }
                    else{
                        colonne2[i] = j;
                    }
                }
            }
        }

        // pour toutes les paires de lignes
        for(int i1 = 0; i1<9; i1++){
            for(int i2 = 0; i2<9; i2++){
                // si elles remplissent les conditions :
                // - différentes
                // - avec deux indices chacune
                // - avec leurs deux indices sur la mêm colonne
                // alors on applique la technique
                if((i1!=i2) && (countl[i1] ==2) && (countl[i2]==2)){
                    if((colonne1[i1] == colonne1[i2]) && (colonne2[i1] ==
colonne2[i2])){
                        for(int i = 0; i<9; i++){
                            // si on ne modifie rien, il ne faut pas renvoyer
true !
                            if((i != i1) && (i != i2) && ((g->notes[i]
[colonne1[i1]][n-1] == true) || (g->notes[i][colonne2[i1]][n-1] == true))){
                                g->notes[i][colonne1[i1]][n-1] = false ;
                                g->notes[i][colonne2[i1]][n-1] = false ;
                                //printf("Technique : x_wing\n");
                                //printf("On retire %d ligne %d et colonnes %d
et %d\n", n, i, colonne1[i1], colonne2[i1]);
                                if (trouve == false){
                                    trouve = true ;
                                }
                            }
                        }
                    }
                }

            }
        }

        // compteur des occurences de n dans les notes dans chaque colonne
        int countc[9] = {0,0,0,0,0,0,0,0,0};

        // indices de deux lignes contenat des indices (utile si count[i] = 2)
        int ligne1[9] = {-1, -1, -1, -1, -1, -1, -1, -1, -1};
        int ligne2[9] = {-1, -1, -1, -1, -1, -1, -1, -1, -1};

        // on parcourt les colonnes
        for(int j = 0; j<9; j++){

            // on parcourt la colonne choisie
            for(int i = 0; i<9; i++){
                // si la case ij peut accueillir n (Attention au décalage
d'indices !)
                if (g->notes[i][j][n-1]){
                    countc[j] ++ ;
                    if(ligne1[j]==-1){
                        ligne1[j] = i;
                    }
                    else{
                        ligne2[j] = i;
                    }
                }
            }
        }
        // pour toutes les paires de colonnes
        for(int j1 = 0; j1<9; j1++){
            for(int j2 = 0; j2<9; j2++){
                // si elles remplissent les conditions :
                // - différentes
                // - avec deux indices chacune
                // - avec leurs deux indices sur la même ligne
                // alors on applique la technique
                if((j1!=j2) && (countc[j1] == 2) && (countc[j2]==2)){
                    if((ligne1[j1] == ligne1[j2]) && (ligne2[j1] ==
ligne2[j2])){
```

```
                        for(int j = 0; j<9; j++){
                            // si on ne modifie rien, il ne faut pas renvoyer
true !
                            if((j != j1) && (j != j2) && ((g->notes[ligne1[j1]]
[j][n-1] == true) || (g->notes[ligne2[j1]][j][n-1] == true))){
                                g->notes[ligne1[j1]][j][n-1] = false ;
                                g->notes[ligne2[j1]][j][n-1] = false ;
                                //printf("Technique : x_wing\n");
                                //printf("On retire %d lignes %d et %d  et
colonne %d\n", n, ligne1[j1], ligne2[j1], j);
                                if (trouve == false){
                                    trouve = true ;
                                }
                            }
                        }
                    }
                }

            }
        }
    }
    return trouve ;
}
```

Fichier mon_code/y_wing.c

```
/*fonction y-wing*/
#include <assert.h>
#include <stdbool.h>
#include<stdio.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

bool same_zone(int i1, int j1, int i2, int j2) ;
// même ligne ou même colonne ou même sous-groupe

int compteur(bool *tab) {
  int compt = 0;
  for (int i = 0; i < 9; i++) {
    if (tab[i]) {
      compt++;
    }
  }
  return compt;
}

bool y_wing(grid_t g) {
  bool verif = false;
  for (int i = 0; i < 9; i++) {   // boucle itérant sur les lignes
    for (int j = 0; j < 9; j++) { // boucle itérant sur les colonnes
      // on parcours chaque case de la grille
      if (compteur(g->notes[i][j]) == 2) {
        // on compte le nombre d'indice par case, si il y en a deux on la
        // considère comme pivot
        int pivot[2] = {-1, -1};
        int pincer_1[2] = {-1, -1};
        int pincer_2[2] = {-1, -1};
        int position_pince[4] = {
            -1, -1, -1, -1}; // tableau contenant l'indice de ligne et colonne
des
                             // pinces (la position du pivot est i,j)
        bool valide[3] = {false, false,
        false}; // ce tableau contient trois booléen indiquant si le
                // pivot et les deux pinces sont valides ou non
        for (int k = 0; k < 9; k++) {
          // première valeur du pivot
          if (g->notes[i][j][k] && pivot[0] == -1) {
            pivot[0] = k;
          }
          // seconde valeur du pivot  (on ne remplit la seconde que si la
          // première est remplie)
          else {
            if (g->notes[i][j][k]) {
              pivot[1] = k;
              valide[0] = true; // le pivot est rempli, on modifie donc le
                                // tableau de booléens
              //printf("On a trouvé un pivot : %d %d\n", i+1, j+1);
            }
          }
        }
        // On cherche la première pince (une valeur en commun avec le pivot)
        for (int i1 = 0; i1 < 9; i1++) {
          for (int j1 = 0; j1 < 9; j1++) {
            if ((i1 != i || j1 != j) && (compteur(g->notes[i1][j1]) == 2) &&
                same_zone(i, j, i1, j1) ) {
              // la case est différente du pivot et le compteur est de deux et
              // la case est en relation avec le pivot, on regarde donc les
              // notes
              pincer_1[0] = -1 ;
              pincer_1[1] = -1 ;
```

```
            for (int k = 0; k < 9; k++) {
              if (g->notes[i1][j1][k] && (k == pivot[0])) {
                pincer_1[0] = k;
              } else {
                if (g->notes[i1][j1][k]) {
                  pincer_1[1] = k;
                }
              }
            }
            if (pincer_1[0] != -1 && pincer_1[1] != -1) {
              valide[1] = true;
              position_pince[0] = i1;
              position_pince[1] = j1;
              //printf("On a trouvé une premiere pince\n");
              // On cherche la seconde pince (une valeur en commun avec le
pivot)
              for (int i2 = 0; i2 < 9; i2++) {
                for (int j2 = 0; j2 < 9; j2++) {
                  if ((i2 != i || j2 != j) && (compteur(g->notes[i2][j2]) ==
2) &&
                      same_zone(i, j, i2, j2) &&
                      !(same_zone(position_pince[0], position_pince[1], i2,
j2))) {
                    pincer_2[0] = -1 ;
                    pincer_2[1] = -1 ;
                    // la case est différente du pivot et de la première
pince, le
                    // compteur est de deux et la case est en relation avec
le pivot,
                    // on regarde donc les notes
                    for (int k = 0; k < 9; k++) {
                      if (g->notes[i2][j2][k] && (k == pivot[1])) {
                        pincer_2[0] = k;
                      } else {
                        if (g->notes[i2][j2][k]) {
                          pincer_2[1] = k;
                        }
                      }
                    }
                    if (pincer_2[0] != -1 && pincer_2[1] != -1) {
                      valide[2] = true;
                      position_pince[2] = i2;
                      position_pince[3] = j2;
                      //printf("On a trouvé une deuxieme pince\n");

                      // on regarde si les deux pinces sont remplies
                      if (valide[1] && valide[2]) {
                        //printf("Coucou1\n");
                        // les premières valeurs doivent être différentes et
les secondes
                        // identique, les deux pinces ne doivent pas être en
relation directe
                        //printf("%d %d %d %d %d %d %d %d\n", pincer_1[0] ,
pincer_2[0], pincer_1[1] , pincer_2[1], position_pince[0],
position_pince[1],position_pince[2], position_pince[3]);
                        if ((pincer_1[0] != pincer_2[0]) && (pincer_1[1] ==
pincer_2[1]) &&
                            !(same_zone(position_pince[0], position_pince[1],
                                        position_pince[2],
position_pince[3]))) {
                          //printf("Coucou2\n");
                          // les pinces sont valides, on doit donc chercher à
élaguer dans
                          // toutes les cases sous l'influence des DEUX
pinces
                          /*ça pourrait être intéréssant d'utiliser les
fonctions same(line,
                           * column, zone) et d'avoir un tableau stockant
l'adresse du pivot
                           * est des pinces (le pivot c'est déjà i et j )*/
                          // on cherche les indices en relation est les deux
pinces donc on
                          // parcours les lignes et colonnes associé à chaque
pince en
                          // vérifiant si la case est en relation avec
l'autre pince
                          for (int i3 = 0; i3 < 9; i3++) {
                            for (int j3 = 0; j3 < 9; j3++) {
                              if (g->notes[i3][j3][pincer_1[1]] &&
                                  same_zone(i3, j3, position_pince[0],
position_pince[1]) &&
                                  same_zone(i3, j3, position_pince[2],
position_pince[3])) {
                                //printf("Coucou3\n");
                                // la cellule est en contact avec les deux
pinces et la notes
                                // commune des pince est présente
                                if ((i3 != position_pince[0] || j3 !=
position_pince[1]) &&
                                    (i3 != position_pince[2] || j3 !=
position_pince[3]) &&
                                    (i3 != i || j3 != j)) {
                                  // les indices ne sont pas ceux d'une pince
ou du pivot,
```

```
                                    // c'est donc valide
                                    verif = true;
                                    // //printf("Y-Wing detecte :) \n");
                                    // //printf("Pivot en %d, %d avec comme
valeurs %d et %d\n",
                                    //   (i + 1), (j + 1), pivot[0]+1,
pivot[1]+1);
                                    // //printf("pince 1 : %d, %d avec %d %d
\n",
                                    //   position_pince[0] + 1,
position_pince[1] + 1,
                                    //   pincer_1[0]+1, pincer_1[1]+1);
                                    // //printf("pince 2 : %d, %d avec %d
%d\n",
                                    //   position_pince[2] + 1,
position_pince[3] + 1,
                                    //   pincer_2[0]+1, pincer_2[1]+1);
                                    // //printf("elaguage de %d en %d, %d\n",
pincer_1[1]+1,i3 + 1, j3 + 1);
                                    g->notes[i3][j3][pincer_1[1]] = false;
                                  }
                                }
                              }
                            }
                            if (verif) {
                              return true;
                            }
                          }
                        }
                      }
                    }
                  }
                }
              }
            }
          }
        }


      }
    }
  }
  //printf("Pas de Y-wing :( \n");
  return false;
}
```

Fichier mon_code/hiddenSingle.c

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

void updateNotes(grid_t g, int row, int col);
void printGrid(int** grid);
void consequences_new_number(int **grid, bool ***notes, int i, int j, float*
nb_techniques);

/*Fonction hidden Single
Entrée : une grille et les notes associées
Sortie : Booléen
Si une note n'est présente qu'une seule fois dans une ligne, colonne ou zones.
Alors elle ne peut être ailleurs et aucune autre notes de peut être dans sa
case */
int hiddenSingle(grid_t g){
    // renvoie true si on parvient à placer un chiffre avec la technique du
hidden single et place ledit chiffre
    // renvoie false sinon

    for(int i = 0; i <9; i++){ // parcourt les lignes
        //on stocke les valeurs possibles sur la ligne
        bool possibilities[9] = {true, true, true, true, true, true, true,
true, true};
        int nb_pos = 9;
        for (int j = 0; j < 9; j++){
            //si la valeur est présente sur la ligne, on met sa valeur de
possibilities à false
            if(g->grid[i][j]!= 0){
                possibilities[g->grid[i][j]-1] = false ;
                nb_pos -- ;
            }
        }

    //on cherche pour chaque valeur non présente si la note est unique
        for(int value = 1; value <= 9; value ++){
            if(possibilities[value-1]){
                int count = 0 ;
```

```
                int col ;
                for(int j = 0; j<9; j++){
                    //lorsque la notes est présente sur une ligne, on itère le
compteur
                    if(g->notes[i][j][value - 1]){
                        count ++;
                        col = j ;
                    }
                }
                if (count == 1){ // si un seul emplacement est disponible alors
on place l'indice
                    g->grid[i][col] = value ;
                    //printf("Technique : hiddenSingle1\n");
                    //printf("row = %d, col = %d, val = %d\n", i, col, value);
                    //consequences_new_number(grid, notes, i, col,
nb_techniques);
                    updateNotes(g, i, col); //on renouvelle la grille de notes
après avoir placé l'indice
                    assert(nb_pos>=1);
                    return nb_pos;
                }
            }
        }
    }
//les boucles suivantes fonctionnes sur le même principes à la différences près
que les boucles itèrent sur les colonnes et les zones
    for(int j = 0; j <9; j++){ // parcourt les colonnes
        bool possibilities[9] = {true, true, true, true, true, true, true,
true, true};
        int nb_pos = 9;
        for (int i = 0; i < 9; i++){
            if(g->grid[i][j]!= 0){
                possibilities[g->grid[i][j]-1] = false ;
                nb_pos--;
            }
        }

        for(int value = 1; value <= 9; value ++){
            if(possibilities[value-1]){
                int count = 0 ;
                int row ;
                for(int i = 0; i<9; i++){
                    if(g->notes[i][j][value - 1]){
                        count ++;
                        row = i ;
                    }
                }
                if (count == 1){ // si un seul emplacement est disponible
                    g->grid[row][j] = value ;
                    //printf("Technique : hiddenSingle2\n");
                    //printf("row = %d, col = %d, val = %d\n", row, j, value);
                    updateNotes(g, row, j);
                    //consequences_new_number(grid, notes, row, j,
nb_techniques);
                    assert(nb_pos>=1);
                    return nb_pos;
                }
            }
        }
    }

    for(int z = 0; z <9; z++){ // parcourt les zones
        bool possibilities[9] = {true, true, true, true, true, true, true,
true, true};
        int nb_pos = 9;
        for (int i = 0; i < 3; i++){
            for(int j = 0; j<3; j++){
                if(g->grid[3*(z/3)+i][3*(z%3) + j]!= 0){
                    possibilities[g->grid[3*(z/3)+i][3*(z%3) + j] - 1] =
false ;
                    nb_pos -- ;
                }
            }
        }

        for(int value = 1; value <= 9; value ++){
            if(possibilities[value-1]){
                int count = 0 ;
                int row ;
                int col ;
                for(int i = 0; i<3; i++){
                    for(int j = 0; j<3 ; j++){
                        if(g->notes[3*(z/3)+i][3*(z%3) + j][value - 1]){
                            count ++;
                            row = 3*(z/3) + i ;
                            col = 3*(z%3) + j ;
                        }
                    }
                }
                if (count == 1){ // si un seul emplacement est disponible
                    g->grid[row][col] = value ;
                    //printf("Technique : hiddenSingle3\n");
                    //printf("row = %d, col = %d, val = %d\n", row, col,
value);
```

```
                    updateNotes(g, row, col);
                    //consequences_new_number(grid, notes, row, col,
nb_techniques);
                    assert(nb_pos>=1);
                    return nb_pos;
                }
            }
        }
    }
    return false ;
}



// itère hiddenSingle autour d'une seule case, pour une seul valeur
int hiddenSingle_one_cell(int** grid, bool*** notes, int i, int j, int k,
float* nb_techniques){
    // renvoie true si on parvient à placer un chiffre avec la technique du
hidden single et place ledit chiffre
    // renvoie false sinon
    //printf("hiddenSingle_one_cell : i = %d, j = %d, k = %d\n", i, j, k);

    //on stocke les valeurs possibles sur la ligne
    bool possible = true;
    int nb_pos = 9;
    for (int j1 = 0; j1 < 9; j1++){
        //si la valeur est présente sur la ligne, on met sa valeur de
possibilities à false
        if(grid[i][j1]== k+1){
            possible=false ;
            nb_pos-- ;
        }
    }
//on cherche pour chaque valeur non présente si la note est unique

    if(possible){
        int count = 0 ;
        int col ;
        for(int j1 = 0; j1<9; j1++){
            //lorsque la note est présente sur une ligne, on itère le compteur
            if(notes[i][j1][k]){
                count ++;
                col = j1 ;
            }
        }
        if (count == 1){ // si un seul emplacement est disponible alors on
place l'indice
            //printf("Technique : hiddenSingle1_one_cell\n");
            //printf("row = %d, col = %d, val = %d\n", i, col, k+1);
            grid[i][col] = k+1 ;
            //printGrid(grid);
            //updateNotes(grid, notes, i, col); //on renouvelle la grille de
notes après avoir placé l'indice
            //consequences_new_number(grid, notes, i, col, nb_techniques);
            return nb_pos;
        }
    }



//les boucles suivantes fonctionnes sur le même principes à la différences près
que les boucles itèrent sur les colonnes et les zones

    bool possibilities1[9] = {true, true, true, true, true, true, true, true,
true};
    nb_pos = 9;
    for (int i1 = 0; i1 < 9; i1++){
        if(grid[i1][j]!= 0){
            possibilities1[grid[i1][j]-1] = false ;
            nb_pos -- ;
        }
    }

    //for(int value = 1; value <= 9; value ++){
        if(possibilities1[k]){
            int count = 0 ;
            int row ;
            for(int i1 = 0; i1<9; i1++){
                if(notes[i1][j][k]){
                    count ++;
                    row = i1 ;
                }
            }
            if (count == 1){ // si un seul emplacement est disponible

                //printf("Technique : hiddenSingle2_one_cell\n");
                //printf("row = %d, col = %d, val = %d\n", row, j, k+1);
                grid[row][j] = k+1 ;
                //printGrid(grid);
                //updateNotes(grid, notes, row, j);
                //consequences_new_number(grid, notes, row, j, nb_techniques);
                return nb_pos;
            }
        }
```

```
    //}


    int z = (3*(i/3)) + j/3 ;
    //printf("z = %d\n", z);
    bool possibilities2[9] = {true, true, true, true, true, true, true, true,
true};
    nb_pos = 9;
    for (int i1 = 0; i1 < 3; i1++){
        for(int j1 = 0; j1<3; j1++){
            if(grid[3*(z/3)+i1][3*(z%3) + j1]!= 0){
                possibilities2[grid[3*(z/3)+i1][3*(z%3) + j1] - 1] = false ;
                nb_pos -- ;
            }
        }
    }

    //for(int k = 1; k <= 9; k ++){
        if(possibilities2[k]){
            int count = 0 ;
            int row ;
            int col ;
            for(int i1 = 0; i1<3; i1++){
                for(int j1 = 0; j1<3 ; j1++){
                    if(notes[3*(z/3)+i1][3*(z%3) + j1][k]){
                        count ++;
                        row = 3*(z/3) + i1 ;
                        col = 3*(z%3) + j1 ;
                    }
                }
            }
            if (count == 1){ // si un seul emplacement est disponible

                //printf("Technique : hiddenSingle3_one_cell\n");
                //printf("row = %d, col = %d, val = %d\n", row, col, k+1);
                grid[row][col] = k+1 ;
                //printGrid(grid);
                //updateNotes(grid, notes, row, col);
                //consequences_new_number(grid, notes, row, col,
nb_techniques);
                return nb_pos;
            }
        }
    //}

    return false ;
}
```

Fichier mon_code/hiddenTriple.c

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

/*Fonction globale hidden triple
Entrée : une grille de note
Sortie : booléen (true si un nouveau triplet caché a été trouvé, false sinon)
Effet de bord : les notes pouvant être retirées grâce au triplet ont été
retirées
(Fonction globale appelant succéssivement hidden triple sur les lignes,
les colonnes et les zones) */
bool hiddenTriple_line(bool ***notes);
bool hiddenTriple_column(bool ***notes);
bool hiddenTriple_zone(bool ***notes);
void free_zones(bool*** zones){
  for(int i = 0; i<9; i++){
    free(zones[i]);
  }
  free(zones);
}

bool hiddenTriple(grid_t g) {
  //printf("Coucou from hiddenTriple\n");
  bool ok;
  //printf("On lance sur les lignes\n");
  ok = hiddenTriple_line(g->notes);
  if (!ok) {
    //printf("On lance sur les colonnes\n");
    ok = hiddenTriple_column(g->notes);
  }
  if (!ok) {
    //printf("On lance sur les zones\n");
    ok = hiddenTriple_zone(g->notes);
  }
  return ok;
}
```

```c
/*fonction hiddenTriple (uniquement sur les lignes)
Entrée : une grille munie de notes
Sortie : un booléen (si un nouveau triplet caché a été trouvé ou non) */
bool hiddenTriple_line(bool ***notes) {
  for (int i = 0; i < 9; i++) {
    int compteur[9] = {0, 0, 0, 0, 0,
                       0, 0, 0, 0}; // tableau stockant le nombre d'occurences
                                    // des valeurs dans les notes (initialisé à
                                    // zéro à chaque nouvelle ligne étudiée)
    for (int valeur = 0; valeur < 9; valeur++) {
      for (int j = 0; j < 9; j++) {
        if (notes[i][j][valeur]) {
          compteur[valeur]++;
        }
      }
    }
    // le tableau est rempli, on peut donc faire une étude exhaustive des
triplets
    // possibles
    for (int j = 0; j < 9; j++) {
      if (compteur[j] == 2 || compteur[j] == 3) {
        // Une note n'est présente que dans deux ou trois cases
        // (le cas une case a déjà été traité par le hiddenSingle,
        // mais une note dasn deux cases peut faire partie d'un hiddenTriple
        // sans faire partie d'un hiddenPair),
        // on examine donc si elle fait partie d'un triplet caché
        for (int k = j + 1; k < 9;
             k++) { // on recherche un triplet parmis les valeurs suivante (la
                    // relation de triplet étant symetrique)
          if (compteur[k] == 2 || compteur[k] == 3) {
            for (int p = k + 1; p < 9; p++) {
              if (compteur[p] == 2 || compteur[p] == 3) {
                // La troisième valeur est aussi présente deux fois, on va
donc
                // regarder leurs places. Une valeur ne pouvant être présente
                // qu'une seule fois dans une case, si le nombre totale de
                // case occupé par ces notes est de trois alors la paire ets
                // conforme
                bool place[9] = {
                    false, false, false, false, false,
                    false, false, false, false}; // on utilise ce tableau pour
                                                 // enregistrer les positions
                                                 // des triplets potentiels
                int count = 0;
                for (int l = 0; l < 9; l++) {
                  if (notes[i][l][k] || notes[i][l][j] || notes[i][l][p]) {
                    place[l] = true;
                    count++;
                  }
                }
                if (count ==
                    3) { // le triple est conforme (trois valeurs différente et
                         // 3 cases), on peut donc élaguer
                  // un Hidden Triple déjà élaguée est identifié comme une
                  // hidden pair, on rajoute donc un booléen qui signale
lorsque
                  // l'on élague afin de renvoyer true
                  bool verif = false;
                  for (int n = 0; n < 9; n++) {
                    if (place[n]) {
                      for (int m = 0; m < 9; m++) {
                        if (m != k && m != j && m != p && notes[i][n][m]) {
                          verif = true; // on a trouvé une note différente du
                                        // triplet dans une des trois cases, on
la
                                        // met alors à false
                          notes[i][n][m] = false;
                        }
                      }
                    }
                  }
                  // si verif est faux c'est que le triplet est déjà élagué, on
en
                  // cherche alors un autre dans le tableau;
                  if (verif) {
                    //printf("Technique : hiddenTriple ligne\n");
                    //printf("%d, %d et %d forme un triplet en ligne %d \n", j
+ 1, k + 1, p + 1, i + 1);
                    return true;
                  }
                }
              }
            }
          }
        }
      }
    }
  }
  // tout les lignes on été parcourues
  //printf("Grille parcourue en entier\n");
  return false;
}
```

```c
// flemme de recopier les commentaires
bool hiddenTriple_column(bool ***notes) {
  for (int j = 0; j < 9; j++) {
    int compteur[9] = {0, 0, 0, 0, 0,
                       0, 0, 0, 0}; // tableau stockant le nombre d'occurence
                                    // des valeurs dans les notes (initialisé à
                                    // zéro à chaque nouvelle colonne étudiée)
    for (int valeur = 0; valeur < 9; valeur++) {
      for (int i = 0; i < 9; i++) {
        if (notes[i][j][valeur]) {
          compteur[valeur]++;
        }
      }
    }
    // le tableau est rempli, on peut donc faire une étude exhaustive des
    // triplets possible
    for (int i = 0; i < 9; i++) {
      if (compteur[i] == 3 || compteur[i] == 2) {
        for (int k = i + 1; k < 9; k++) {
          // on recherche une paire parmis les valeurs suivante (la
          // relation de paire étant symetrique)
          if (compteur[k] == 3 || compteur[k] == 2) {
            // la seconde valeur est aussi présente trois
            // fois, on va donc
            // regarder leurs places une valeur ne pouvant être présente
            // qu'une seule fois dans une case, si le nombre totale de
            // case occupé par ces notes est de deux alors la paire ets
            // conforme
            for (int p = k + 1; p < 9; p++) {
              if (compteur[p] == 3 || compteur[p] == 2) {
                bool place[9] = {
                    false, false, false, false, false,
                    false, false, false, false};
                // on utilise ce tableau pour enregistrer les positions
                // des triplets potentiels
                int count = 0;
                for (int l = 0; l < 9; l++) {
                  if (notes[l][j][i] || notes[l][j][k] || notes[l][j][p]) {
                    place[l] = true;
                    count++;
                  }
                }
                if (count == 3) {
                  // le triple est conforme (trois valeurs différente et
                  // 3 cases), on peut donc élaguer
                  // un Hidden Triple déjà élaguée est identifié comme une
                  // hidden pair, on rajoute donc un booléen qui signale
lorsque
                  // l'on élague afin de renvoyer true
                  bool verif = false;
                  for (int n = 0; n < 9; n++) {
                    if (place[n]) {
                      for (int m = 0; m < 9; m++) {
                        if (m != k && m != i && m != p && notes[n][j][m]) {
                          verif = true; // on a trouvé une note différente de
la
                                        // paire dans une des deux cases, on la
                                        // met alors à false
                          notes[n][j][m] = false;
                        }
                      }
                    }
                  }
                  // si verif est faux c'est que la paire est déjà élagué, on
en
                  // cherche alors une autre dans le tableau;
                  if (verif) {
                    //printf("Technique : hiddenTriple colonne\n");
                    //printf("%d, %d et %d forme un triplet en colonne %d \n",
i + 1, k + 1, p + 1, j + 1);
                    return true;
                  }
                }
              }
            }
          }
        }
      }
    }
  }
  // tout les lignes on été parcouru
  //printf("Grille parcourue en entier\n");
  return false;
}

bool hiddenTriple_zone(bool ***notes) {
  // conversion de la grille en zones
  bool ***zones = malloc(9 * sizeof(bool **));
  // on construit une grille répartis en zones :
  // plus simple pour cette étude
  // même morceau de fonction que last_remaining_cell_zone
  assert(zones != NULL);
  for (int i = 0; i < 9; i++) {
    zones[i] = malloc(9 * sizeof(bool *));
```

```
    assert(zones[i] != NULL);
    for (int j = 0; j < 9; j++) {
      zones[i][j] = notes[3*(i/3) + j/3][3*(i%3) + j%3] ;

    }
  }

  for (int i = 0; i < 9; i++) {
    int compteur[9] = {0, 0, 0, 0, 0,
                       0, 0, 0, 0}; // tableau stockant le nombre d'occurence
                                    // des valeurs dans les notes (initialisé à
                                    // zéro à chaque nouvelle ligne étudié)
    for (int valeur = 0; valeur < 9; valeur++) {
      for (int j = 0; j < 9; j++) {
        if (zones[i][j][valeur]) {
          compteur[valeur]++;
        }
      }
    }
    // le tableau est rempli, on peut donc faire une étude exhaustive des pairs
    // possible
    for (int j = 0; j < 9; j++) {
      if (compteur[j] == 3) {
        for (int k = j + 1; k < 9;
             k++) { // on recherche une paire parmis les valeurs suivante (la
                    // relation de paire étant symetrique)
          if (compteur[k] ==
              3) { // la seconde valeur est aussi présente deux fois, on va
donc
                   // regarder leurs places une valeur ne pouvant être présente
                   // qu'une seule fois dans une case, si le nombre totale de
                   // case occupé par ces notes est de deux alors la paire ets
                   // conforme
            for (int p = k + 1; p < 9; p++) {
              if (compteur[p] == 3) {
                bool place[9] = {
                    false, false, false, false, false,
                    false, false, false, false}; // on utilise ce tableau pour
                                                 // enregistrer les positions
                                                 // des paires potentielles
                int count = 0;
                for (int l = 0; l < 9; l++) {
                  if (zones[i][l][k] || zones[i][l][j] || zones[i][l][p]) {
                    place[l] = true;
                    count++;
                  }
                }
                if (count ==
                    3) { // le triple est conforme (trois valeurs différente et
                         // 3 cases), on peut donc élaguer
                  // un Hidden Triple déjà élaguée est identifié comme une
                  // hidden pair, on rajoute donc un booléen qui signale
lorsque
                  // l'on élague afin de renvoyer true
                  bool verif = false;
                  for (int n = 0; n < 9; n++) {
                    if (place[n]) {
                      for (int m = 0; m < 9; m++) {
                        if (m != k && m != j && m != p && zones[i][n][m]) {
                          verif = true; // on a trouvé une note différente de
la
                                        // paire dans une des deux cases, on la
                                        // met alors à false
                          zones[i][n][m] = false;
                          notes[3 * (i / 3) + n / 3][3 * (i % 3) + n % 3][m] =
                              false;
                        }
                      }
                    }
                  }
                  // si verif est faux c'est que la paire est déjà élagué, on
en
                  // cherche alors une autre dans le tableau;
                  if (verif) {
                    //printf("Technique : hiddenTriple zone\n");
                    //printf("%d, %d et %d forme un triplet en zone %d \n", j +
1, k + 1, p + 1, i + 1);
                    free_zones(zones);
                    return true;
                  }
                }
              }
            }
          }
        }
      }
    }
  }
  // tout les lignes on été parcouru
  //printf("Grille parcourue en entier\n");
  free_zones(zones);
  return false;
}
```

Fichier mon_code/initialize_notes.c

```c
#include <assert.h>
#include <stdbool.h>
#include<stdlib.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

bool ***createNotes();
void updateNotes(grid_t g, int row, int col);

void initialize_notes(grid_t g){
    g->notes = createNotes();
    assert(g->notes!=NULL);
    // notes[i][j][k] == true si la case ij peut accueillir l'indice k+1

    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            updateNotes(g, i, j);
        }
    }
    assert(g->notes!=NULL);

}
```

Fichier mon_code/lastRemainingCell.c

```c
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

bool lastRemainingCell_row(int **grid);
bool lastRemainingCell_column(int **grid);
bool lastRemainingCell_zone(int **grid);

/*Fonction verif colonne / lignes
Entrée : une grille, une valeur recherché et la ligne ou colonne dans laquelle
elle est recherché
Sortie : si la valeur est présente ou non*/

bool verif_ligne(int valeur, int **grille, int l) {
    for (int j = 0; j < 9; j++) {
        if (valeur == grille[l][j]) {
            return true;
        }
    }
    return false;
}

bool verif_colonne(int valeur, int **grille, int c) {
    for (int i = 0; i < 9; i++) {
        if (valeur == grille[i][c]) {
            return true;
        }
    }
    return false;
}

bool verif_zone(int valeur, int **grille, int row_zone, int col_zone){
    for(int i = 0; i<3; i++){ //itération sur les lignes et colonnes de la zone
        for(int j = 0; j<3; j++){
            if (valeur == grille[row_zone + i][col_zone + j]){
                return true ;
            }
        }
    }
    return false ;
}


// on réunit les trois
bool lastRemainingCell(grid_t g) {

    bool ok = false;
    ok = lastRemainingCell_column(g->grid);
    if (!ok) {
        ok = lastRemainingCell_row(g->grid);
        if (!ok) {
            ok = lastRemainingCell_zone(g->grid);
        }
    }
    return ok;
}
```

```c
/*fonction last remaining cell sur les lignes */
bool lastRemainingCell_row(int **grid) {
    // //printf("Technique essayee : dernière case restante ligne\n");
    for (int i = 0; i < 9; i++) {
        int possibilities[9] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        // on reinitialise les chiffres possibles pour chaque ligne

        // si un chiffre est déjà présent dans la ligne, on l'exclut des possibilités
        for (int j = 0; j < 9; j++) {
            if (grid[i][j] != 0) {
                possibilities[grid[i][j] - 1] = 0;
                // on ne testera donc que les valeur non présentes
            }
        }

        for (int chiffre = 1; chiffre < 10; chiffre++) {
            if (possibilities[chiffre - 1] != 0) {

                bool cell[9] = {true, true, true, true, true, true, true, true, true};
                // pour chaque chiffre, on cree un tableau des cases disponibles
                // dans la ligne pour ce chiffre

                // si une case est pleine, alors elle n'est pas disponible
                for (int j = 0; j < 9; j++) {
                    if (grid[i][j] != 0) {
                        cell[j] = false;
                    }
                }

                // si une des règles empêche le chiffre d'être sur une case,
                // on retire cette case de cell, le tableau des cases possibles
                for(int j =0; j<9; j++){
                    if (cell[j] && (verif_zone(chiffre, grid, 3*(i/3), 3*(j/3)) || verif_colonne(chiffre, grid, j))) {
                        cell[j] = false;
                    }

                }

                int count = 0;
                for (int i = 0; i < 9; i++) {
                    // on compte le nombre de place possible pour cette valeur
                    if (cell[i]) {
                        count++;
                    }
                }

                if (count == 1) {
                    // il y a une unique place pour cette valeur on peut donc la placer
                    int k = 0;
                    while (k < 9 && !cell[k]) {
                        k++;
                    }
                    grid[i][k] = chiffre;
                    // mise à jour des grilles : une valeur a été placée
                    //printf("Technique : Last Remaining Cell Row\n");
                    //printf("row = %d, col = %d, val = %d\n", i, k, chiffre);
                    return true;
                }
            }
        }
    }
    return false;
}


bool lastRemainingCell_column(int **grid) {

    for (int j = 0; j < 9; j++) {
        // //printf("attaque de la colonne %d \n", j);


        int possibilities[9] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        // on reinitialise les possibilités pour chaque lignes
        for (int i = 0; i < 9; i++) {
            if (grid[i][j] != 0) {
                possibilities[grid[i][j] - 1] = 0; // on ne filtre donc que les valeur non présentes
            }
        }

        for (int chiffre = 1; chiffre <= 9; chiffre++) {
            if (possibilities[chiffre - 1] != 0) {
                // si le chiffre n'est pas déjà présent sur la colonne

                bool cell[9] = {true, true, true, true, true, true, true, true, true};
                // pour chaque chiffre, on cree un tableau des cases disponibles
```

```
                // si la case n'est pas vide, alors elle n'est pas disponible
                for (int i = 0; i < 9; i++) {
                    if (grid[i][j] != 0) {
                        cell[i] = false;
                    }
                }

                // si la case ne peut pas accueillir ce chiffre à cause d'une
des règles,
                // alors on la retire des cases disponibles
                for (int i = 0; i < 9; i++) {
                    if (verif_zone(chiffre, grid, 3*(i/3) , 3*(j/3)) ||
verif_ligne(chiffre, grid, i)) {
                        cell[i] = false;
                    }
                }

                int count = 0;
                for (int k = 0; k < 9; k++) {
                    // on compte le nombre de place possible pour cette valeur
                    if (cell[k]) {
                        count++;
                    }
                }

                if (count == 1) {
                    // il y a une unique place pour cette valeur on peut donc
la placer
                    int k = 0;
                    while (k < 9 && !cell[k]) {
                        k++;
                    }
                    grid[k][j] = chiffre; // mise à jour des grilles : une
valeur a été placée
                    //printf("Technique : Last Remaining Cell Column\n");
                    //printf("row = %d, col = %d, val = %d\n", k, j, chiffre);
                    return true;
                }
            }
        }
    }
    return false;
}


bool lastRemainingCell_zone(int **grid) {
    // la grille sera de 9 par 9 (tableau de dimension 2)
    int **zones = malloc(9 * sizeof(int *));
    // on construit une grille répartis en zones :
    // plus simple pour cette étude
    assert(zones != NULL);
    for (int i = 0; i < 9; i++) {
        zones[i] = malloc(9 * sizeof(int));
        assert(zones[i] != NULL);
    }

    for (int z = 0; z < 9; z++) {
        for (int c = 0; c < 9; c++) {
          zones[z][c] = grid[3*(z/3) + c / 3][3 * (z % 3) + c % 3];
        }
    }

    for (int z = 0; z < 9; z++) {
        // pour chaque zone de la grille

        int possibilities[9] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        // on reinitialise le tableau des chiffres possibles pour chaque zone

        for (int j = 0; j < 9; j++) { // on trouve les places disponibles
            if (zones[z][j] != 0) {
                possibilities[zones[z][j]-1] = 0;
                // il ne restera que les valeurs non présentes
            }
        }


        for (int chiffre = 1; chiffre <= 9; chiffre++){

            if (possibilities[chiffre - 1] != 0) {
                // le chiffre n'est pas présent dans la zone : on cherche alors
sa
                // place

                bool cell[9] = {true, true, true, true, true, true, true, true,
true};
                // cell : tableau des cellules possibles, pour un chiffre donné

                // Si une case est pleine, elle ne peut pas accueillir le
chiffre
                for (int j = 0; j < 9; j++) {
                    if (zones[z][j] != 0) {
                        cell[j] = false;
                    }
```

```
            }


            for (int j = 0; j < 9; j++) { // on cherche les cases vides
                if (cell[j]) {
                    // si  la case est libre
                    // appel à une fonction auxiliaire verif de ligne ou
colonne
                    // voisines
                    for (int i = 0; i < 3; i++) {
                        if (verif_ligne(possibilities[chiffre - 1], grid, 3
* (z / 3) + i)) {

                            for (int k = 0; k < 3; k++) {
                                cell[3 * i + k] = false;
                                // on retire les cases présentes dans cette
                                // ligne du tableau des possibilités;
                            }
                        }
                        if (verif_colonne(possibilities[chiffre - 1], grid,
3 * (z % 3) + i)) {

                            for (int k = 0; k < 3; k++) {
                                cell[i + 3 * k] = false;
                                // on retire les cases présentes dans cette
                                // colonne du tableau des possibilités;
                            }
                        }
                    }
                }
            }

            int count = 0;
            for (int i = 0; i < 9; i++) {
                // on compte le nombre de place possible pour cette valeur
                if (cell[i]) {
                    count++;
                }
            }
            if (count == 1) {
                // il y a une unique place pour cette valeur on peut donc
la placer

                int k = 0;
                while (k < 9 && !cell[k]) {
                    k++;
                }
                // mise à jour des grilles : une valeur a été placée
                grid[3*(z/3) + k / 3][3 * (z % 3) + k % 3] = chiffre;
                //printf("Technique : Last Remaining Cell zone\n");
                //printf("Row = %d , col = %d, val = %d \n", 3*(z/3) + k/3,
3*(z%3) + k%3, chiffre);
                return true;
            }
        }

    }
    return false;
}
```

Fichier mon_code/last_free_cell.c

```
# include <assert.h>
# include <stdbool.h>
#include <stdio.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

void printGrid(int** grid);

//void consequences_new_number(int **grid, bool ***notes, int i, int j, float*
nb_techniques);
bool est_ok(int** grid);

bool lastFreeCell(grid_t g){ // OK !
    // Entrée : une grille ( tableau de 9*9 entiers)
    // Sortie : la grille modifiée sur une unique case si c'est possible avec
cette technique et true , false sinon
    // Technique du LastFreeCell : si une zone possède déjà 8 indices, remplit
le 9ème
    if(!est_ok(g->grid)){
        return false;
    }

    int sum ; // pour ne pas avoir besoin de rechercher la valeur manquante
    int counter ; // je compte les cases pleines
    for(int row = 0; row<9 ; row ++ ){ // test des lignes
        sum = 45 ;
        counter = 0; // compte les indices présents
        for(int col = 0; col<9; col ++){
```

```
            if (g->grid[row][col] != 0){
                counter ++ ;
                sum -= g->grid[row][col];
            }
        }
        if (counter==8){ // si 8 cases sont pleines :
            for (int col = 0; col<9; col ++ ){ // recherche l'emplacement
libre
                if(g->grid[row][col] == 0){
                    assert(sum > 0);
                    assert(sum<=9);
                    g->grid[row][col] = sum ; // sum vaut alors la valeur
manquante
                    //printf("Technique : lastFreeCell\n");
                    //printf("row = %d, col = %d, val = %d\n", row, col, g-
>grid[row][col]);
                    //consequences_new_number(grid, notes, row, col,
nb_techniques);
                    return true;
                }
            }
        }
    }

    for(int col = 0; col<9 ; col ++ ){ //idem sur les colonnes
        sum = 45 ;
        counter = 0;
        for(int row = 0; row<9; row ++){
            if (g->grid[row][col] != 0){
                counter ++ ;
                sum -= g->grid[row][col];
            }
        }
        if (counter==8){
            for (int row = 0; row<9; row ++ ){
                if(g->grid[row][col] == 0){
                    assert(sum > 0);
                    assert(sum<=9);
                    g->grid[row][col] = sum ;
                    //printf("Technique : lastFreeCell\n");
                    //printf("row = %d, col = %d, val = %d\n", row, col, g-
>grid[row][col]);
                    //consequences_new_number(g->grid, g->notes, row, col,  g-
>nb_techniques);
                    return true;
                }
            }
        }
    }

    int colGroup ;
    int rowGroup ;
    for (int group = 0; group<9; group ++){//idem sur les blocs : petite
complication sur les indices, mais sinon c'est identique
        sum = 45 ;
        counter = 0;
        colGroup = group % 3 ;
        rowGroup = group / 3 ;
        for(int row = 0; row <3; row++){
            for(int col = 0; col < 3; col++){
                if (g->grid[3*rowGroup + row] [3*colGroup + col] != 0){
                    counter ++ ;
                    sum -= g->grid[3*rowGroup + row] [3*colGroup + col];
                }
            }
        }
        if (counter==8){
            for(int row = 0; row <3; row ++){
                for(int col = 0; col <3; col ++){
                    if (g->grid[3*rowGroup + row] [3*colGroup + col] == 0){
                        assert(sum > 0);
                        assert(sum<=9);
                        g->grid[3*rowGroup + row] [3*colGroup + col] = sum ;
                        //printf("Technique : lastFreeCell\n");
                        //printf("row = %d, col = %d, val = %d\n", (3*rowGroup
+ row), (3*colGroup + col), g->grid[3*rowGroup + row] [3*colGroup + col]);
                        //consequences_new_number(g->grid, g->notes, 3*rowGroup
+ row, 3*colGroup + col,  g->nb_techniques);
                        return true ;
                    }
                }
            }
        }

    }
    return false ;
}

bool lastFreeCell_one_cell(grid_t g, int i, int j){
    //printf("LastFreeCell_one_cell\n");
    //printf("i = %d, j = %d\n", i, j);
    if(!est_ok(g->grid)){
        return false;
    }
```

```
    int sum ; // pour ne pas avoir besoin de rechercher la valeur manquante
    int counter ; // je compte les cases pleines
    sum = 45 ;
    counter = 0; // compte les indices présents
    for(int col = 0; col<9; col ++){
        if (g->grid[i][col] != 0){
            counter ++ ;
            sum -= g->grid[i][col];
        }
    }
    if (counter==8){ // si 8 cases sont pleines :
        for (int col = 0; col<9; col ++ ){ // recherche l'emplacement libre
            if(g->grid[i][col] == 0){
                assert(sum > 0);
                assert(sum<=9);
                g->grid[i][col] = sum ; // sum vaut alors la valeur manquante
                //printf("Technique : lastFreeCell_one_cell\n");
                //printf("row = %d, col = %d, val = %d\n", i, col, g->grid[i]
[col]);
                //consequences_new_number(grid, notes, i, col,  nb_techniques);
                return true;
            }
        }
    }

    sum = 45 ;
    counter = 0;
    for(int row = 0; row<9; row ++){
        if (g->grid[row][j] != 0){
            counter ++ ;
            sum -= g->grid[row][j];
        }
    }
    if (counter==8){
        for (int row = 0; row<9; row ++ ){
            if(g->grid[row][j] == 0){
                assert(sum > 0);
                assert(sum<=9);
                g->grid[row][j] = sum ;
                //printf("Technique : lastFreeCell_one_cell\n");
                //printf("row = %d, col = %d, val = %d\n", row, j, g->grid[row]
[j]);
                //consequences_new_number(grid, notes, row, j, nb_techniques);
                return true;
            }
        }
    }

    sum = 45 ;
    counter = 0;
    int colGroup = i / 3 ;
    int rowGroup = j /3 ;
    for(int row = 0; row <3; row++){
        for(int col = 0; col < 3; col++){
            if (g->grid[3*rowGroup + row] [3*colGroup + col] != 0){
                counter ++ ;
                sum -= g->grid[3*rowGroup + row] [3*colGroup + col];
            }
        }
    }
    if (counter==8){
        for(int row = 0; row <3; row ++){
            for(int col = 0; col <3; col ++){
                if (g->grid[3*rowGroup + row] [3*colGroup + col] == 0){
                    assert(sum > 0);
                    assert(sum<=9);
                    g->grid[3*rowGroup + row] [3*colGroup + col] = sum ;
                    //printf("Technique : lastFreeCell_one_cell\n");
                    //printf("row = %d, col = %d, val = %d\n", (3*rowGroup +
row), (3*colGroup + col), g->grid[3*rowGroup + row] [3*colGroup + col]);
                    //consequences_new_number(grid, notes, 3*rowGroup + row,
3*colGroup + col, nb_techniques);
                    return true ;
                }
            }
        }
    }
    return false ;
}
```

Fichier mon_code/last_possible_number.c

```
# include <assert.h>
# include <stdbool.h>
#include<stdio.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;
```

```c
bool same_zone(int row, int col, int i, int j);

bool lastPossibleNumber(grid_t g){
    // Entrée : une grille ( tableau de 9*9 entiers)
    // Sortie : la grille modifiée sur une unique case si c'est possible avec
cette technique et true , false sinon
    int nPossibilities = 9 ; // compteur
    int sum = 45 ; // pour faire le décompte, ça m'évite de devoir chercher
l'indice restant à nouveau
    for(int row = 0; row<9; row++){
        for(int col = 0; col<9; col++){// je parcours la grille
            if (g->grid[row][col]==0){ // si la case est vide :
                bool possibilities[9] =
{true,true,true,true,true,true,true,true,true}; // tableau des valeurs
possibles
                nPossibilities = 9 ;
                sum = 45 ;
                for(int i = 0; i<9; i++){  //parcours sur les lignes
                    for(int j = 0; j<9; j++){ //parcours sur les colonnes
                        if (g->grid[i][j] != 0 && (i!=row || col != j) &&
same_zone(row, col, i, j)){
                            if (possibilities[g->grid[i][j]-1]){ // Rien à
faire si la valeur est déjà impossible !
                                possibilities[g->grid[i][j]-1] = false ;
                                nPossibilities--;
                                sum -= g->grid[i][j];
                            }
                        }
                    }
                }
                if (nPossibilities == 1 ){ // si il ne me reste qu'une seule
possibilité
                    g->grid[row][col]=sum; // sum vaut alors la seule valeur
restante
                    //printf("Technique : last Possible Number\n");
                    //printf("row = %d, col = %d, val = %d\n", row, col,
sum); //tests
                    return true ;
                }
            }
        }
    }
    return false ; // pas trouvé !
}
```

Fichier mon_code/lecture.c

```c
/*Fichier pour la lecture de la base de donnée */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

// n : ligne à lire
//le compteur sera mis à jour dans le main
int** lecture(int n, char* nom_de_la_base){
    FILE* f ;
    f = fopen(nom_de_la_base, "r");

    // on va à la ligne souhaitée
    int line = 1;
    char c;
    while (line < n){
        c = fgetc(f);
        if (c == '\n'){
            line++;
        }
    }

    int** grid = malloc(9*sizeof(int*));
    assert(grid!=NULL);
    for(int i =0; i<9; i++){
        grid[i] = malloc(9*sizeof(int));
        assert(grid[i]!=NULL);
    }

    for(int i = 0; i<9; i++){
        for(int j = 0; j<9; j++){
            grid[i][j] = fgetc(f) - '0';
            // correctif ascii -> chiffre
        }
    }
    fclose(f);
    return grid ;

}
```

Fichier mon_code/lecture_db_B.c

```c
/*Fichier pour la lecture de la base de données avec difficultés */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct {
  int difficulty ;
  int** grid ;
```

```
} grid_one_diff ;

// n : ligne à lire
//le compteur sera mis à jour dans le main
grid_one_diff lecture_db_B(int n, char* nom_de_la_base, int cap){
    FILE* f ;
    f = fopen(nom_de_la_base, "r");


    // on va à la ligne souhaitée
    int line = 1;
    char c;
    while (line < n){
        while (fgetc(f) != ','); // on saute la source

        for(int i = 0; i<9; i++){
            for(int j = 0; j<9; j++){
                fgetc(f) ;
            }
        }
        fgetc(f); // on saute la virgule
        while (fgetc(f) != ','); // on saute la solution
        int diff;
        fscanf(f, "%d", &diff);
        if(diff<cap ){
            line++;
        }
        while(fgetc(f) != '\n');
    }

    grid_one_diff g ;

    int** mygrid = malloc(9*sizeof(int*));
    assert(mygrid!=NULL);
    for(int i =0; i<9; i++){
        mygrid[i] = malloc(9*sizeof(int));
        assert(mygrid[i]!=NULL);
    }
    g.difficulty = 42069 ;
    while(g.difficulty>=cap){
        while (fgetc(f) != ','); // on saute la source

        for(int i = 0; i<9; i++){
            for(int j = 0; j<9; j++){
                char d = fgetc(f) ;
                if (d == '.'){
                    d = '0' ;
                }
                mygrid[i][j] = d - '0';
                // correctif ascii -> chiffre
            }
        }
        fgetc(f); // on saute la virgule
        while (fgetc(f) != ','); // on saute la solution

        fscanf(f, "%d", &g.difficulty);
        g.grid = mygrid;

        while(fgetc(f) != '\n');
    }
    fclose(f);



    return g ;

}
```

Fichier mon_code/lecture_db_C.c

```
/*Fichier pour la lecture de la base de données avec difficultés */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct {
  int difficulty ;
  int** grid ;
} grid_one_diff ;

// n : ligne à lire
//le compteur sera mis à jour dans le main
grid_one_diff lecture_db_C(int n, char* nom_de_la_base, int cap){
    FILE* f ;
    f = fopen(nom_de_la_base, "r");


    // on va à la ligne souhaitée
    int line = 1;
    char c;
    while (line < n){
        while (fgetc(f) != ','); // on saute l'identifiant

        for(int i = 0; i<9; i++){
```

```
        for(int j = 0; j<9; j++){
            fgetc(f) ;
        }
    }
    fgetc(f); // on saute la virgule
    while (fgetc(f) != ','); // on saute le nombre de joueurs
    int diff;
    fscanf(f, "%d", &diff);
    if(diff<cap ){
        line++;
    }
    while(fgetc(f) != '\n');
}

grid_one_diff g ;

int** mygrid = malloc(9*sizeof(int*));
assert(mygrid!=NULL);
for(int i =0; i<9; i++){
    mygrid[i] = malloc(9*sizeof(int));
    assert(mygrid[i]!=NULL);
}
g.difficulty = 42069 ;
while(g.difficulty>=cap){
    while (fgetc(f) != ','); // on saute l'identifiant

    for(int i = 0; i<9; i++){
        for(int j = 0; j<9; j++){
            char d = fgetc(f) ;
            if (d == '.'){
                d = '0' ;
            }
            mygrid[i][j] = d - '0';
            // correctif ascii -> chiffre
        }
    }
    fgetc(f); // on saute la virgule
    while (fgetc(f) != ','); // on saute le nombre de joueurs

    fscanf(f, "%d", &g.difficulty);
    g.grid = mygrid;

    while(fgetc(f) != '\n');
}
fclose(f);
```

```
    return g ;

}
```

Fichier mon_code/lecture_db_diff.c

```
/*Fichier pour la lecture de la base de données avec difficultés */
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct {
  float D_TO ;
  float D_TR ;
  int** grid ;
} grid_diffs ;

// n : ligne à lire
//le compteur sera mis à jour dans le main
grid_diffs lecture_db_diffs(int n, char* nom_de_la_base, int cap){
    FILE* f ;
    f = fopen(nom_de_la_base, "r");

    // on va à la ligne souhaitée
    int line = 1;
    char c;
    while (line < n){
        c = fgetc(f);
        if (c == '\n'){

            line++;



        }
    }
    float d_tr = cap+1 ;
    float d_to ;
    grid_diffs g ;

    int** mygrid = malloc(9*sizeof(int*));
    assert(mygrid!=NULL);
    for(int i =0; i<9; i++){
        mygrid[i] = malloc(9*sizeof(int));
        assert(mygrid[i]!=NULL);
    }
    while(d_tr >cap){
```

```
        while (fgetc(f) != ',');

        for(int i = 0; i<9; i++){
            for(int j = 0; j<9; j++){
                char d = fgetc(f) ;
                if (d == '.'){
                    d = '0' ;
                }
                mygrid[i][j] = d - '0';
                // correctif ascii -> chiffre
            }
        }
        fgetc(f); // on saute la virgule

        fscanf(f, "%f", &d_to);
        fgetc(f);

        fscanf(f, "%f", &d_tr);
        g.grid = mygrid;
        g.D_TO = d_to ;
        g.D_TR = d_tr ;
        while(fgetc(f) != '\n');
    }
    fclose(f);


    return g ;

}
```

Fichier mon_code/main.c

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

int group(int i, int j) { return 3 * (i / 3) + j / 3; } // ok
// renvoie le numéro du sous-groupe (de 0 à 8)
```

```
bool same_zone(int i1, int j1, int i2, int j2) { // ok
    // i1, i2, j1, j2 entre 0 et 8
    return (i1 == i2) || (j1 == j2) || (group(i1, j1) == group(i2, j2));
} // même ligne ou même colonne ou même sous-groupe

void printGrid(int **grid); // ok

bool lastFreeCell(grid_t g);
bool lastRemainingCell(grid_t g);
bool lastPossibleNumber(grid_t g);

void updateNotes(grid_t g, int row, int col);
void initialize_notes(grid_t g);
void free_grid(int **grid);
void free_notes(bool ***notes);

bool hiddenSingle(int **grid, bool ***notes, float *nb_tech);
bool nakedSingle(int **grid, bool ***notes, float *nb_tech);

bool hiddenPair(bool ***notes);
bool nakedPair(bool ***notes);

bool hiddenTriple(bool ***notes);
bool nakedTriple(bool ***notes);

bool pointingPair(bool ***notes);
bool boxLineReduction(bool ***notes);

bool x_wing(bool ***notes);
bool y_wing(bool ***notes);
bool swordfish(bool ***notes);

bool solve(int **grid);
bool solve_notes(grid_t g, bool(**techniques)(grid_t g), int n);
bool backtrack(int **grid, bool ***notes, float *nb_techniques);
void solve_simple_notes_backtrack(grid_t g, bool(**techniques)(grid_t g), int
n);

int **lecture(int n, char *nom_de_la_base);
typedef struct {
    float D_TO;
    float D_TR;
    int **grid;
} grid_diffs;
grid_diffs lecture_db_diffs(int n, char *nom_de_la_base, int cap);

typedef struct {
```

```
    int difficulty;
    int **grid;
} grid_one_diff;
grid_one_diff lecture_db_B(int n, char *nom_de_la_base, int cap);
grid_one_diff lecture_db_C(int n, char *nom_de_la_base, int cap);



int **grid_of_string(char *s);


void updateNotes(grid_t g, int row, int col);

void print_notes(bool ***notes);

typedef struct var_s{
    /* Une variable est de la forme p_i,j,k, elle indique si la case i,j
contient k */
    /* 0 <= i,j < 9 */
    /* 1 <= k <= 9 */
    int i;
    int j;
    int k;
}var;

typedef struct {
    /* On représente un litéral par une variable (un entier)
    et une positivite (1 si litéral positif, 0 si litéral négatif)
    On représente donc une clause par un tableau de litéraux */
    int nb_lit ;
    var* vars ;
    bool* positif ;
}clause;


typedef struct {
    /* Ce type est beaucoup moins général ; le filtre est spécifique à cet
usage */
    int nb_var ;
    var* vars ;
    bool* filtre ;
}clause_1in9;


struct k_cnf_s {
    int m ; //nb_clauses
    int k ;
    clause* clauses ;
};
typedef struct k_cnf_s* k_cnf;

k_cnf sudoku_to_cnf(int** grid);
void print_k_cnf(k_cnf f);
float assess_cnf(int** grid);

float assess_techniques(int** grid, float* coeffs, float* coeffs_first_use);
int assess_nb_notes(int** grid);
float assess_repartition(int** grid);
float assess_repartition_valeurs(int** grid);
int assess_nb_clues(int** grid);

void solve_cnf(k_cnf f, var(*h)(k_cnf), int* nb_disjonctions, int* nb_quines);
void free_k_cnf(k_cnf f);
/* Random */
var heuristique_0(k_cnf f);
/* Minimum */
var heuristique_1(k_cnf f);
/* Maximum */
var heuristique_2(k_cnf f);

void print_tab_int(int *tab, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d  ", tab[i]);
    }
    printf("\n");
}
void print_tab_float(float *tab, int size) {
    for (int i = 0; i < size; i++) {
        printf("%.4f,  ", tab[i]);
    }
    printf("\n");
}

float calcule_cout(float *coeffs, float *coeffs_first_use, float **results,
float *difficulties, int results_size);

void calcule_coeffs(float* coeffs, float* coeffs_first_use, float** results,
float* difficulties, int results_size);
void calcule_coeffs_neg(float* coeffs, float* coeffs_first_use, float**
results, float* difficulties, int results_size);
float *cree_coeffs() {
    /* Tableau des techniques :
    0 - last Free Cell
    1 - nakedSingle
```

```
    2 - hiddenSingle
    3 - nakedPair
    4 - nakedTriple
    5 - hiddenPair
    6 - hiddenTriple
    7 - pointing Pair / Triple
    8 - Box line reduction
    9 - X-Wing
    10 - Y-wing
    11 - Swordfish
    12 - Backtracking
    */
    float *coeffs = malloc(13 * sizeof(float));
    assert(coeffs != NULL);
    for(int i = 0; i<13; i++){
        coeffs[i] = (rand() % 100 + 10) / 50. ;
    }
    return coeffs;
}

float *cree_coeffs_first_use() {
    float *coeffs_first_use = malloc(13 * sizeof(float));
    assert(coeffs_first_use != NULL);
    for(int i = 0; i<13; i++){
        //coeffs_first_use[i] = (rand() % 100 + 10) / 50. ;
        coeffs_first_use[i] = 0;
    }
    //coeffs_first_use[2] = 0;
    //coeffs_first_use[0] = 0;
    return coeffs_first_use;
}



void test_heuristiques(){
    FILE *h = fopen("results_heuristics/db_B.txt", "w");
    int results_size = 200 ;

    for(int nbGrille = 0; nbGrille<results_size; nbGrille++){
        grid_one_diff g1 = lecture_db_B(nbGrille+2, "grilles/db_B.csv", 5);
        k_cnf phi = sudoku_to_cnf(g1.grid);
        printGrid(g1.grid);
        print_k_cnf(phi);

        int profondeur_max = 42;

        int* nb_disjonctions = malloc(profondeur_max*sizeof(int));
        assert(nb_disjonctions!=NULL);
        int* nb_quines = malloc(profondeur_max*sizeof(int));
        assert(nb_quines!=NULL);
        for(int i = 0; i<profondeur_max; i++){
            nb_disjonctions[i] = 0;
            nb_quines[i] = 0;
        }

        solve_cnf(phi, &heuristique_1, nb_disjonctions, nb_quines);
        print_tab_int(nb_disjonctions, profondeur_max);
        print_tab_int(nb_quines, profondeur_max);

        for(int i = 0; i<profondeur_max; i++){
            fprintf(h, "%d,  ",nb_disjonctions[i]);
        }
        for(int i = 0; i<profondeur_max; i++){
            fprintf(h, "%d,  ",nb_quines[i]);
        }
        fprintf(h,"%d, ", g1.difficulty);
        if(nbGrille<results_size-1){
            fprintf(h,";\n");
        }

        free_k_cnf(phi);
        free(nb_disjonctions);
        free(nb_quines);
        free_grid(g1.grid);
    }
    fclose(h);
}

void test_techniques(){
    int n = 12 ;
    bool(**techniques)(grid_t) = malloc(12*sizeof(bool(*)(grid_t)));
    assert(techniques!=NULL);
    techniques[0] = &lastFreeCell ;
    techniques[1] = (bool(*)(grid_t))&nakedSingle ;
    techniques[2] = (bool(*)(grid_t))&hiddenSingle ;
    techniques[3] = (bool(*)(grid_t))&pointingPair ;
    techniques[4] = (bool(*)(grid_t))&nakedPair ;
    techniques[5] = (bool(*)(grid_t))&hiddenPair ;
    techniques[6] = (bool(*)(grid_t))&nakedTriple ;
    techniques[7] = (bool(*)(grid_t))hiddenTriple ;
    techniques[8] = (bool(*)(grid_t))&boxLineReduction ;
    techniques[9] = (bool(*)(grid_t))x_wing ;
    techniques[10] = (bool(*)(grid_t))y_wing ;
    techniques[11] = (bool(*)(grid_t))swordfish ;
```

```c
srand(time(NULL));
float *coeffs = cree_coeffs();
float *coeffs_first_use = cree_coeffs_first_use();

int results_size = 1000 ;
FILE *f = fopen("resultats.txt", "w");


float **results = malloc(results_size * sizeof(float *));
assert(results != NULL);
float *difficulties = malloc(results_size * sizeof(float));
assert(difficulties != NULL);

for (int nbGrille = 0; nbGrille < results_size ; nbGrille++) {
    if(nbGrille%100 == 0){
        printf("Grille n %d\n", nbGrille);
    }
    grid_one_diff g = lecture_db_B(nbGrille+2, "grilles/db_B.csv", 10);
    //int** g2 = lecture(nbGrille, "grilles/top50000.txt");

    //printGrid(g.grid);
    grid_t g2 = malloc(sizeof(struct grid_s));
    assert(g2!=NULL);
    g2->grid = g.grid ;
    difficulties[nbGrille] = (float) g.difficulty;
    g2->nb_techniques = malloc(13 * sizeof(float));
    assert(g2->nb_techniques!=NULL);
    for (int i = 0; i < 13; i++) {
        g2->nb_techniques[i] = 0.;
    }

    solve_simple_notes_backtrack(g2, techniques, 12);
    if(nbGrille%100 == 0){
        print_tab_float(g2->nb_techniques, 13);
    }
    // print_tab_int(nb_tech, 10);
    results[nbGrille] = g2->nb_techniques;
    for (int i = 0; i < 13; i++) {
        fprintf(f, "%f, ", g2->nb_techniques[i]);
    }

    fprintf(f, "%f ;\n", difficulties[nbGrille]);

    free_grid(g.grid);
    free(g2);
}

for(int i = 0; i<13; i++){
    int count = 0 ;
    int n = 0;
    for(int j = 0; j<results_size; j++){
        count += (int) results[j][i];
        if (results[j][i]>0.01){
            n ++;
        }
    }
    printf("%d, %d \n", count, n);
}
printf("\n");


calcule_coeffs_neg(coeffs,coeffs_first_use,results,difficulties,
results_size);

/* Calcul des coefficients par descente de gradient au formalisme douteux
*/
for (int i = 0; i < 13; i++) {
    fprintf(f, "%f, ", coeffs[i]);
}
fprintf(f,"42;\n");
for (int i = 0; i < 13; i++) {
    fprintf(f, "%f, ", coeffs_first_use[i]);
}
fprintf(f,"42\n");

fclose(f);

free(coeffs);
free(coeffs_first_use);
for(int i = 0; i<results_size; i++){
    free(results[i]);
}
free(results);
free(difficulties);
free(techniques);
}

void test_criteres(float* coeffs, float* coeffs_first_use){
    FILE* g = fopen("results_db_0/results_criteria.txt", "w");
    int results_size = 344 ;
    for(int i = 0; i<results_size; i++){
        /* La résolution altère la grille donnée en argument*/
```

```
        //printf("################## Grille n° %d ############\n",i);
        grid_diffs g1 = lecture_db_diffs(i+2, "grilles/
Base_de_donnees_evaluees.csv", 42);
        int nb_clues = assess_nb_clues(g1.grid);

        float f2 = assess_cnf(g1.grid);
        int nb_notes = assess_nb_notes(g1.grid);
        float repartition = assess_repartition(g1.grid);
        float repartition_valeurs = assess_repartition_valeurs(g1.grid);
        float f1 = assess_techniques(g1.grid, coeffs, coeffs_first_use);
        fprintf(g, "%f , %f , %f , %d, %d , %f, %f ",g1.D_TR, f1, f2,nb_clues,
nb_notes, repartition, repartition_valeurs);
        if(i<results_size-1){
            fprintf(g,";\n");
        }
    }
    fclose(g);
}

int main() {

    test_heuristiques();

    //test_techniques() ;

    // test_criteres();

    return 0;
}
```

Fichier mon_code/Makefile

```
all: main
    ./main

CC = gcc
CFLAGS = -lm -g -pg -fsanitize=address -O2

SRCS = $(wildcard *.c)
HEADERS = $(wildcard *.h)

main: $(SRCS)
    $(CC) $(CFLAGS) $(SRCS) -o "$@" -g
main-debug: $(SRCS)
    $(CC) $(CFLAGS) -O0 $(SRCS) -o "$@"

clean:
    rm -f main main-debug
```

Fichier mon_code/moindres_carres.py

```
import numpy as np
import matplotlib.pyplot as plt

np.set_printoptions(threshold=1000000)

### Lit resultats.txt
f = open("resultats.txt","r")
texte = f.read()
m = np.matrix(texte)
f.close()


n,p = m.shape




######## Sélectionne les données utiles
m = np.delete(m,range(n-2,n),0)
y = np.delete(m,range(p-1),1)
#m = np.delete(m,p-1,1)



n,p = m.shape

# autorise un coeff constant
for i in range(n) :
    m[i,p-1] = 1



print("n =", n)




""" Ancienne version
# Supprime les colonnes liés, de telle sorte que m soit de rang maximal
#print(m)
j = 0
for k in range(p) :
```

```
    if np.linalg.matrix_rank(np.delete(m, range(j+1,p),1)) == j+1 :
        j+=1
    else :
        print("colonne ",k," supprimée")

        m = np.delete(m,j,1)
    n,p = m.shape
#print("m = ",m)
#print("y = ",y)

n,p = m.shape



#### Résout le système
mt = np.linalg.matrix_transpose(m)
mtm = mt*m
#print(mtm)
#print(mt*y)
x = np.linalg.solve(mt*m,mt*y)
print("x=",x)
"""
mt = np.linalg.matrix_transpose(m)
x = np.linalg.lstsq(mt*m,mt*y)[0]
#print("x = ", np.linalg.matrix_transpose(x))
for i in range(p) :
    print("%.3f  " %x[i,0], end = "")

# Calcule l'écart-type
variance = 0
d = n * [0]
for i in range(n) :
    diff = 0
    for j in range(p) :
        diff += m[i,j] * x[j,0]
    d[i] = diff
    variance += (diff - y[i])**2
variance /= n
print("variance = ",variance)
#print("ecart-type = ",np.sqrt(variance))

######## Calcule la corrélation #######

dcmoy = np.average(d)
ddmoy = np.average(y)
corr = 0
for i in range(n) :
    corr += (d[i] - dcmoy)*(y[i]-ddmoy)

corr/= (np.std(y) * np.std(d) * n)

print("correlation = ",corr)

y2 = n*[42]
for i in range(n):
    y2[i] = y[i,0]
identite = range(n)



### Tri
t = []
for i in range(n):
    t.append((y2[i],d[i]))
t.sort()
for i in range(n):
    d[i] = t[i][1]
    y2[i] = t[i][0]

dc_per_dd = []
for i in range(10) :
    dc_per_dd.append([])
means = 10 * [42]
std = 10 * [42]
for i in range(n):
    dc_per_dd[int(y2[i])].append( d[i])

for i in range(10) :
    means[i] = np.average(dc_per_dd[i])
    std[i] = np.std(dc_per_dd[i])



### Affichage

# plt.scatter(identite,d, label = "Difficulté calculée")
# plt.scatter(identite,y2, label = "Difficulté donnée")
# plt.xlabel("Sudokus")
# plt.ylabel("Difficulté(réel arbitraire)")
plt.scatter(y2,d, s = 20, label = "Corrélation = %f" %(np.array(corr)[0][0]))
plt.errorbar(range(10), means, std,  marker = "s",  color = "orange", label =
"Moyennes et écarts-types")
plt.xlabel("Difficulté donnée par la base")
plt.ylabel("Difficulté calculée")
```

```
#plt.title("Méthode analytique")

plt.legend()

plt.show()



########## m2 prend en compte les coeffs constants ############

m = np.delete(m,p-1,1)
n,p = np.shape(m)

#n = n+10
m2 = []
#print(m)
for i in range(n) :
    m2.append((2*p)*[0])
    for j in range(p) :
        m2[i][j] = m[i,j]
        if m[i,j] > 0.01 :
            m2[i][j+p] = 1
# transforme le tableau de tableaux en matrice (c'est pas la mme chose
uurg)
m2 = np.matrix(m2)
#n= n-10

## Cette procéudre fait le même travail que les lignes suivantes, je l'ai
découverte après
m2t =np.linalg.matrix_transpose(m2)
x2 = np.linalg.lstsq(m2t*m2,m2t*y)[0]
#print("x2 = ",np.linalg.matrix_transpose(x2))
print("X2 = ")
for i in range(p) :
    print("%.3f  " %x2[i,0], end = "")
print("")
for i in range(p) :
    print("%.3f  " %x2[i+p,0], end = "")
print("")

n,p2 = m2.shape
""" ancienne version :
# Supprime les colonnes liés, de telle sorte que m soit de rang maximal
j = 0


for k in range(p2) :
    if np.linalg.matrix_rank(np.delete(m2, range(j+1,p2),1)) == j+1 :
        j+=1
    else :
        print("colonne ",k," supprimée")

        m2 = np.delete(m2,j,1)
    n,p2 = m2.shape

#print(m2)
m2t = np.linalg.matrix_transpose(m2)
#print(m2t*m2)
#print(m2t*y)

#### m2^Tm2 est inversible grace au code précédent
x2 = np.linalg.solve(m2t*m2,m2t*y)
print("x2 = ",x2)

"""



cout = 0
d = n * [0]
for i in range(n) :
    diff = 0
    for j in range(p2) :
        if j < p :
            diff += m2[i,j]*x2[j]
        elif m2[i,j]>0.01 :
            diff += x2[j]
    d[i] = diff
    cout += (diff - y[i])**2
cout /= n
print("variance = ",cout)




### Tri et affichage
y2 = n*[42]
for i in range(n):
    y2[i] = y[i,0]
identite = range(n)

t = []
for i in range(n):
```

```
    t.append((y2[i],d[i]))
t.sort()
for i in range(n):
    d[i] = t[i][1]
    y2[i] = t[i][0]

dc_per_dd = []
for i in range(10) :
    dc_per_dd.append([])
means = 10 * [42]
std = 10 * [42]
for i in range(n-2):
    dc_per_dd[int(y2[i])].append( d[i])

for i in range(10) :
    means[i] = np.average(dc_per_dd[i])
    std[i] = np.std(dc_per_dd[i])

######## Calcule la corrélation #######

dcmoy = np.average(d)
ddmoy = np.average(y2)
corr = 0
for i in range(n) :
    corr += (d[i] - dcmoy)*(y2[i]-ddmoy)

corr/= (np.std(y2) * np.std(d) * n)

print("correlation = ",corr)

#plt.scatter(identite,d, label = "Difficulté calculée")
#plt.scatter(identite,y2, label = "Difficulté donnée")

plt.scatter(y2,d, s = 20,  label = "Corrélation = %f" %(np.array(corr)[0][0]))
plt.errorbar(range(10), means, std,  marker = "s",  color = "orange", label =
"Moyennes et écarts-types")
plt.xlabel("Difficulté donnée par la base")
plt.ylabel("Difficulté calculée")
plt.title("Évaluation avec coefficients de première utilisation")
plt.legend()
plt.show()
```

Fichier mon_code/moindres_carres_log.py

```
import numpy as np
import matplotlib.pyplot as plt

np.set_printoptions(threshold=1000000)

### Lit resultats.txt
f = open("resultats.txt","r")
texte = f.read()
m = np.matrix(texte)
f.close()


n,p = m.shape



######## Sélectionne les données utiles
m = np.delete(m,range(n-2,n),0)
y = np.delete(m,range(p-1),1)
#m = np.delete(m,p-1,1)



n,p = m.shape



# autorise un coeff constant
for i in range(n) :
    m[i,p-1] = 1

#m = np.log(m+1)

print("n =", n)

#y = np.exp(y/2)


for j in range(p) :
    for i in range(n) :
        m[i,j] = m[i,j]**(1/(j+1))


""" Ancienne version
# Supprime les colonnes liés, de telle sorte que m soit de rang maximal
#print(m)
```

```
j = 0
for k in range(p) :
    if np.linalg.matrix_rank(np.delete(m, range(j+1,p),1)) == j+1 :
        j+=1
    else :
        print("colonne ",k," supprimée")

        m = np.delete(m,j,1)
    n,p = m.shape
#print("m = ",m)
#print("y = ",y)

n,p = m.shape



#### Résout le système
mt = np.linalg.matrix_transpose(m)
mtm = mt*m
#print(mtm)
#print(mt*y)
x = np.linalg.solve(mt*m,mt*y)
print("x=",x)
"""
mt = np.linalg.matrix_transpose(m)
x = np.linalg.lstsq(mt*m,mt*y)[0]

for i in range(p) :
    print("%.3f  " %x[i,0], end = "")
#print("x = ", np.linalg.matrix_transpose(x))


# Calcule l'écart-type
variance = 0
d = n * [0]
for i in range(n) :
    diff = 0
    for j in range(p) :
        diff += m[i,j] * x[j,0]
    d[i] = diff
    variance += (diff - y[i])**2
variance /= n
print("variance = ",variance)
#print("ecart-type = ",np.sqrt(variance))

######## Calcule la corrélation #######

dcmoy = np.average(d)
ddmoy = np.average(y)
corr = 0
for i in range(n) :
    corr += (d[i] - dcmoy)*(y[i]-ddmoy)

corr/= (np.std(y) * np.std(d) * n)

print("correlation = ",corr)

y2 = n*[42]
for i in range(n):
    y2[i] = y[i,0]
identite = range(n)



### Tri
t = []
for i in range(n):
    t.append((y2[i],d[i]))
t.sort()
for i in range(n):
    d[i] = t[i][1]
    y2[i] = t[i][0]

# dc_per_dd = []
# for i in range(10) :
#     dc_per_dd.append([])
# means = 10 * [42]
# std = 10 * [42]
# for i in range(n):
#     dc_per_dd[int(y2[i])].append( d[i])

# for i in range(10) :
#     means[i] = np.average(dc_per_dd[i])
#     std[i] = np.std(dc_per_dd[i])



### Affichage

# plt.scatter(identite,d, label = "Difficulté calculée")
# plt.scatter(identite,y2, label = "Difficulté donnée")
# plt.xlabel("Sudokus")
# plt.ylabel("Difficulté(réel arbitraire)")
plt.scatter(y2,d, s = 20, label = "Corrélation = %f" %(np.array(corr)[0][0]))
```

```
#plt.errorbar(range(10), means, std,  marker = "s",  color = "orange", label =
"Moyennes et écarts-types")
plt.xlabel("Difficulté donnée par la base")
plt.ylabel("Difficulté calculée")
#plt.title("Méthode analytique")

plt.legend()

plt.show()



########## m2 prend en compte les coeffs constants ###########

m = np.delete(m,p-1,1)
n,p = np.shape(m)

#n = n+10
m2 = []
#print(m)
for i in range(n) :
    m2.append((2*p)*[0])
    for j in range(p) :
        m2[i][j] = 0
        if m[i,j] > 0.01 :
            m2[i][j+p] = 1
# transforme le tableau de tableaux en matrice (c'est pas la mme chose
uurg)
m2 = np.matrix(m2)
#n= n-10

## Cette procéudre fait le même travail que les lignes suivantes, je l'ai
découverte après
m2t =np.linalg.matrix_transpose(m2)
x2 = np.linalg.lstsq(m2t*m2,m2t*y)[0]
#print("x2 = ",np.linalg.matrix_transpose(x2))
print("X2 = ")
for i in range(p) :
    print("%.3f  " %x2[i,0], end = "")
print("")
for i in range(p) :
    print("%.3f  " %x2[i+p,0], end = "")
print("")

n,p2 = m2.shape
""" ancienne version :
# Supprime les colonnes liés, de telle sorte que m soit de rang maximal
j = 0


for k in range(p2) :
    if np.linalg.matrix_rank(np.delete(m2, range(j+1,p2),1)) == j+1 :
        j+=1
    else :
        print("colonne ",k," supprimée")

        m2 = np.delete(m2,j,1)
    n,p2 = m2.shape

#print(m2)
m2t = np.linalg.matrix_transpose(m2)
#print(m2t*m2)
#print(m2t*y)

#### m2^Tm2 est inversible grace au code précédent
x2 = np.linalg.solve(m2t*m2,m2t*y)
print("x2 = ",x2)

"""



cout = 0
d = n * [0]
for i in range(n) :
    diff = 0
    for j in range(p2) :
        if j < p :
            diff += m2[i,j]*x2[j]
        elif m2[i,j]>0.01 :
            diff += x2[j]
    d[i] = diff
    cout += (diff - y[i])**2
cout /= n
print("variance = ",cout)




### Tri et affichage
y2 = n*[42]
for i in range(n):
    y2[i] = y[i,0]
```

```
identite = range(n)

t = []
for i in range(n):
    t.append((y2[i],d[i]))
t.sort()
for i in range(n):
    d[i] = t[i][1]
    y2[i] = t[i][0]

# dc_per_dd = []
# for i in range(10) :
#     dc_per_dd.append([])
# means = 10 * [42]
# std = 10 * [42]
# for i in range(n-2):
#     dc_per_dd[int(y2[i])].append( d[i])

# for i in range(10) :
#     means[i] = np.average(dc_per_dd[i])
#     std[i] = np.std(dc_per_dd[i])

######## Calcule la corrélation #######

dcmoy = np.average(d)
ddmoy = np.average(y2)
corr = 0
for i in range(n) :
    corr += (d[i] - dcmoy)*(y2[i]-ddmoy)

corr/= (np.std(y2) * np.std(d) * n)

print("correlation = ",corr)

#plt.scatter(identite,d, label = "Difficulté calculée")
#plt.scatter(identite,y2, label = "Difficulté donnée")

plt.scatter(y2,d, s = 20,  label = "Corrélation = %f" %(np.array(corr)[0][0]))
#plt.errorbar(range(10), means, std,  marker = "s",  color = "orange", label =
"Moyennes et écarts-types")
plt.xlabel("Difficulté donnée par la base")
plt.ylabel("Difficulté calculée")
plt.title("Méthode analytique - coefficients de première utilisation")
plt.legend()
plt.show()
```

Fichier mon_code/nakedPair.c

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

/*Fonction globale naked pair
Entrée : une grille de notes
Sortie : booléen (true si une nouvelle paire nue a été trouvée, false sinon )
Effet de bord : les notes pouvant être retirées grâce à la paire ont été
retirées
(Fonction globale appelant successivement naked pair sur les lignes,
les colonnes et les zones )*/
// Ces fonctions sont inspiré du Hidden Pair même si le raisonnement est
inverse
bool nakedPair_line(bool ***notes);
bool nakedPair_column(bool ***notes);
bool nakedPair_zone(bool ***notes);
//float* consequences_removed_note(int** grid, bool*** notes, int i, int j, int
k, float* nb_techniques);
void printGrid(int** grid);
void free_zones(bool*** zones);

bool nakedPair(grid_t g) {
    bool ok  ;
    ok = nakedPair_line(g->notes);
    if (!ok) {
        ok = nakedPair_column(g->notes);
    }
    if (!ok) {
        ok = nakedPair_zone(g->notes);
    }

    return ok;
}

bool nakedPair_zone(bool ***notes) {
    // conversion de la grille en zones
```

```c
    bool ***zones = malloc(9 * sizeof(bool **));
    // on construit une grille répartis en zones :
    // plus simple pour cette étude
    // même morceau de fonction que last_remaining_cell_zone
    assert(zones != NULL);
    for (int i = 0; i < 9; i++) {
        zones[i] = malloc(9 * sizeof(bool *));
        assert(zones[i] != NULL);
        for (int j = 0; j < 9; j++) {
            zones[i][j] = notes[3*(i/3) + j/3][3*(i%3) + j%3] ;
        }
    }

    for (int i = 0; i < 9; i++) {
        int compteur[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0};
        // tableau contenant le nombre de notes d'une case
        for (int cell = 0; cell < 9; cell++) {
            for (int j = 0; j < 9; j++) { // parcours du tableau de booléen en
i, cell
                if (zones[i][cell][j]) {    // note présente
                    compteur[cell]++;
                    // on incrémente le nombre de notes présentes dans cette
case
                }
            }
        }
        for (int j = 0; j < 9; j++) {
            if (compteur[j] == 2) {
            // une case ne contient que deux notes, on examine donc si c'est
une paire nue
                for (int k = j + 1; k < 9; k++) {
                    if (compteur[k] == 2) {
                    // on a trouvé une seconde case avec seulement deux notes,
on compare alors ces notes
                        bool valeur[9] = {false, false, false, false, false,
false, false, false, false};
                        // on utilise ce tableau pour enregistrer les positions
des paires potentielles
                        int count = 0;
                        for (int l = 0; l < 9; l++) {
                            if (zones[i][j][l] || zones[i][k][l]) {
                                valeur[l] = true;
                                count++;
                            }
                        }
                        // il n'y a que deux valeurs différentes dans les deux
cases, c'est
                        // donc une paire
                        if (count == 2) {
                            // une paire nue est présente, on elague si ça
n'est pas déjà fait
                            bool verif = false;
                            for (int n = 0; n < 9; n++) {
                                if (valeur[n]) {
                                    for (int m = 0; m < 9; m++) {
                                        if (m != j && m != k && zones[i][m][n])
{
                                            verif = true;
                                            zones[i][m][n] = false;
                                            notes[3 * (i / 3) + m / 3][3 * (i %
3) + m % 3][n] = false;
                                        }
                                    }
                                }
                            }
                            if (verif) {
                                //printf("Technique : nakedPair zone\n");
                                //printf("les cases %d et %d forme une paire
nue zone %d\n", j + 1, k + 1, i + 1);
                                free_zones(zones);
                                return true;
                            }
                        }
                    }
                }
            }
        }
    }
    //printf("Grille parcourue\n");
    free_zones(zones);
    return false;
}

bool nakedPair_column(bool ***notes) {
    for (int i = 0; i < 9; i++) {
        int compteur[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0};
        // tableau contenant le nombre de note d'une case
        for (int cell = 0; cell < 9; cell++) {
            for (int j = 0; j < 9; j++) { // parcours du tableau de booléen en
i, cell
                if (notes[cell][i][j]) {    // note présente
                    compteur[cell]++; // on incrémente le nombre de notes
présentent dans
                                                                    // cette case
```

```
                }
            }
        }
        for (int j = 0; j < 9; j++) {
            if (compteur[j] == 2) { // une case ne contient que deux notes, on
examine
                                                    // donc si c'est
une paire nue
                for (int k = j + 1; k < 9; k++) {
                    if (compteur[k] == 2) { // on a trouvé une seconde case
avec seulement
                                                            // deux
notes, on compare alors ces notes
                        bool valeur[9] = {
                                false, false, false, false, false,
                                false, false, false, false}; // on utilise ce
tableau pour
                                                                        //
enregistrer les positions des
                                                                        //
paires potentielles
                        int count = 0;
                        for (int l = 0; l < 9; l++) {
                            if (notes[j][i][l] || notes[k][i][l]) {
                                valeur[l] = true;
                                count++;
                            }
                        }
                        // il n'y a que deux valeurs différentes dans les deux
cases, c'est
                        // donc une paire
                        if (count == 2) { // une paire nue est présente, on
elague si ça
                                                    // n'est pas déjà
fait
                            bool verif = false;
                            for (int n = 0; n < 9; n++) {
                                if (valeur[n]) {
                                    for (int m = 0; m < 9; m++) {
                                        if (m != j && m != k && notes[m][i][n])
{

                                            verif = true;
                                            notes[m][i][n] = false;
                                        }
                                    }
                                }
                            }
                            if (verif) {
                                //printf("Technique : nakedPair colonne\n");
                                //printf("les cases %d et %d forme une paire
nue colonne %d\n", j + 1, k + 1, i + 1);
                                return true;
                            }
                        }
                    }
                }
            }
        }
    }
    //printf("Grille parcourue\n");
    return false;
}

bool nakedPair_line(bool ***notes) {
    for (int i = 0; i < 9; i++) {
        int compteur[9] = {
                0, 0, 0, 0, 0,
                0, 0, 0, 0}; // tableau contenant le nombre de note d'une case
        for (int cell = 0; cell < 9; cell++) {
            for (int j = 0; j < 9; j++) { // parcours du tableau de booléen en
i, cell
                if (notes[i][cell][j]) {    // note présente
                    compteur[cell]++; // on incrémente le nombre de notes
présentent dans
                                                            // cette case
                }
            }
        }
        for (int j = 0; j < 9; j++) {
            if (compteur[j] == 2) { // une case ne contient que deux notes, on
examine
                                                                // donc si c'est
une paire nue
                for (int k = j + 1; k < 9; k++) {
                    if (compteur[k] == 2) { // on a trouvé une seconde case
avec seulement
                                                                    // deux
notes, on compare alors ces notes
                        bool valeur[9] = {
                                false, false, false, false, false,
                                false, false, false, false}; // on utilise ce
tableau pour
                                                                            //
enregistrer les positions des
```

```
                                                                //
paires potentielles
                        int count = 0;
                        for (int l = 0; l < 9; l++) {
                            if (notes[i][j][l] || notes[i][k][l]) {
                                valeur[l] = true;
                                count++;
                            }
                        }
                        // il n'y a que deux valeurs différentes dans les deux
cases, c'est
                        // donc une paire
                        if (count == 2) { // une paire nue est présente, on
elague si ça
                                                    // n'est pas déjà
fait
                            bool verif = false;
                            for (int n = 0; n < 9; n++) {
                                if (valeur[n]) {
                                    for (int m = 0; m < 9; m++) {
                                        if (m != j && m != k && notes[i][m][n])
{

                                            verif = true;
                                            notes[i][m][n] = false;
                                        }
                                    }
                                }
                            }
                            if (verif) {
                                //printf("Technique : nakedPair ligne\n");
                                //printf("les cases %d et %d forme une paire
nue ligne %d\n", j + 1, k + 1, i + 1);
                                return true;
                            }
                        }
                    }
                }
            }
        }
    }
    //printf("Grille parcourue\n");
    return false;
}

bool nakedPair_one_zone_one_value(int** grid, bool ***notes, int z, int k,
float* nb_techniques);
bool nakedPair_one_column_one_value(int** grid, bool ***notes, int i, int k,
float* nb_techniques);
bool nakedPair_one_line_one_value(int** grid, bool ***notes, int i, int k,
float* nb_techniques);

bool nakedPair_one_cell_value(grid_t g, int i, int j, int k){
    bool ok ;
    ok = nakedPair_one_zone_one_value(g->grid, g->notes, 3*(i/3) + j/3, k, g-
>nb_techniques);
    if (!ok){
        ok = nakedPair_one_line_one_value(g->grid, g->notes, i, k, g-
>nb_techniques);
    }
    if(!ok){
        ok = nakedPair_one_column_one_value(g->grid, g->notes, j, k, g-
>nb_techniques);
    }
    return ok ;
}



bool nakedPair_one_zone_one_value(int** grid, bool ***notes, int z, int k,
float* nb_techniques) {
    // conversion de la grille en zones
    bool ***zones = malloc(9 * sizeof(bool **));
    // on construit une grille répartis en zones :
    // plus simple pour cette étude
    // même morceau de fonction que last_remaining_cell_zone
    assert(zones != NULL);
    for (int i = 0; i < 9; i++) {
        zones[i] = malloc(9 * sizeof(bool *));
        assert(zones[i] != NULL);
        for (int j = 0; j < 9; j++) {
            zones[i][j] = notes[3*(i/3) + j/3][3*(i%3) + j%3] ;
        }
    }

    int compteur[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0};
    // tableau contenant le nombre de notes d'une case
    for (int cell = 0; cell < 9; cell++) {
        for (int j = 0; j < 9; j++) { // parcours du tableau de booléen en i,
cell
            if (zones[z][cell][j]) {    // note présente
                compteur[cell]++;
                // on incrémente le nombre de notes présentes dans cette case
            }
        }
    }
```

```
    for (int j = 0; j < 9; j++) {
        if (compteur[j] == 2) {
        // une case ne contient que deux notes, on examine donc si c'est une
paire nue
            if (j!= k && compteur[k] == 2) {
            // on a trouvé une seconde case avec seulement deux notes, on
compare alors ces notes
                bool valeur[9] = {false, false, false, false, false, false,
false, false, false};
                // on utilise ce tableau pour enregistrer les positions des
paires potentielles
                int count = 0;
                for (int l = 0; l < 9; l++) {
                    if (zones[z][j][l] || zones[z][k][l]) {
                        valeur[l] = true;
                        count++;
                    }
                }
                // il n'y a que deux valeurs différentes dans les deux cases,
c'est
                // donc une paire
                if (count == 2) {
                    // une paire nue est présente, on elague si ça n'est pas
déjà fait
                    bool verif = false;
                    bool** todo = malloc(9*sizeof(bool*));
                    assert(todo!=NULL);
                    int count = 0;
                    for(int n = 0; n <9; n++){
                        todo[n] = malloc(9*sizeof(bool));
                        for(int m = 0; m<9; m++){
                            todo[n][m] = false;
                        }
                    }
                    for (int n = 0; n < 9; n++) {
                        if (valeur[n]) {
                            for (int m = 0; m < 9; m++) {
                                if (m != j && m != k && zones[z][m][n]) {
                                    verif = true;
                                    zones[z][m][n] = false;
                                    todo[n][m] = true ;
                                    count ++;
                                    notes[3 * (z / 3) + m / 3][3 * (z % 3) + m
% 3][n] = false;
                                }
                            }
                        }
                    }
                    if (verif) {
                        //printf("Technique : nakedPair zone_one_cell\n");
                        //printf("les cases %d et %d forme une paire nue zone
%d\n", j + 1, k + 1, z + 1);
                        // for(int n = 0; n<9; n++){
                        //     for(int m = 0; m<9; m++){
                        //         if(todo[n][m]){
                        //             float * to_add =
consequences_removed_note(grid, notes, z, m, n, nb_techniques);
                        //             for(int i = 0; i<13; i++){
                        //                 nb_techniques[i] += to_add[i] *
count ;
                        //             }
                        //             free(to_add);
                        //         }
                        //     }
                        // }
                        for(int i = 0; i<9; i++){
                            free(todo[i]);
                        }
                        free(todo);
                        free_zones(zones);
                        return true;
                    }
                    for(int i = 0; i<9; i++){
                        free(todo[i]);
                    }
                    free(todo);
                }
            }

        }
    }
    free_zones(zones);
    //printf("Grille parcourue\n");
    return false;
}

bool nakedPair_one_column_one_value(int** grid, bool ***notes, int i, int k,
float* nb_techniques) {

    int compteur[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0};
    // tableau contenant le nombre de note d'une case
    for (int cell = 0; cell < 9; cell++) {
        for (int j = 0; j < 9; j++) { // parcours du tableau de booléen en i,
cell
```

```
            if (notes[cell][i][j]) {    // note présente
                compteur[cell]++;
                // on incrémente le nombre de notes présentes dans cette case
            }
        }
    }
    for (int j = 0; j < 9; j++) {
        if (compteur[j] == 2) { // une case ne contient que deux notes, on
examine
                                                    // donc si c'est une
paire nue

            if (j!= k && compteur[k] == 2) { // on a trouvé une seconde case
avec seulement
                                                        // deux notes, on
compare alors ces notes
                bool valeur[9] = {
                        false, false, false, false, false,
                        false, false, false, false}; // on utilise ce tableau
pour
                                                                    //
enregistrer les positions des
                                                                    //
paires potentielles
                int count = 0;
                for (int l = 0; l < 9; l++) {
                    if (notes[j][i][l] || notes[k][i][l]) {
                        valeur[l] = true;
                        count++;
                    }
                }
                // il n'y a que deux valeurs différentes dans les deux cases,
c'est
                // donc une paire
                if (count == 2) { // une paire nue est présente, on elague si
ça
                                                // n'est pas déjà fait
                    bool verif = false;
                    for (int n = 0; n < 9; n++) {
                        if (valeur[n]) {
                            for (int m = 0; m < 9; m++) {
                                if (m != j && m != k && notes[m][i][n]) {
                                    verif = true;
                                    notes[m][i][n] = false;
                                }
                            }
                        }
                    }
                    if (verif) {
                        //printf("Technique : nakedPair colonne one cell\n");
                        //printf("les cases %d et %d forme une paire nue
colonne %d\n", j + 1, k + 1, i + 1);
                        return true;
                    }
                }
            }

        }
    }

    //printf("Grille parcourue\n");
    return false;
}

bool nakedPair_one_line_one_value(int** grid, bool ***notes, int i, int k,
float* nb_techniques) {

    int compteur[9] = {
            0, 0, 0, 0, 0,
            0, 0, 0, 0}; // tableau contenant le nombre de note d'une case
    for (int cell = 0; cell < 9; cell++) {
        for (int j = 0; j < 9; j++) { // parcours du tableau de booléen en i,
cell
            if (notes[i][cell][j]) {    // note présente
                compteur[cell]++; // on incrémente le nombre de notes
présentent dans
                                                        // cette case
            }
        }
    }
    for (int j = 0; j < 9; j++) {
        if (compteur[j] == 2) {
            // une case ne contient que deux notes, on examine
            // donc si c'est une paire nue
            if (j != k && compteur[k] == 2) {
            // on a trouvé une seconde case avec seulement
            // deux notes, on compare alors ces notes
                bool valeur[9] = {
                        false, false, false, false, false,
                        false, false, false, false};
                // on utilise ce tableau pour enregistrer les positions des
                // paires potentielles
                int count = 0;
                for (int l = 0; l < 9; l++) {
```

```
                if (notes[i][j][l] || notes[i][k][l]) {
                    valeur[l] = true;
                    count++;
                }
            }
            // il n'y a que deux valeurs différentes dans les deux cases,
            // c'est donc une paire
            if (count == 2) {
                // une paire nue est présente, on elague si ça
                // n'est pas déjà fait
                bool verif = false;
                for (int n = 0; n < 9; n++) {
                    if (valeur[n]) {
                        for (int m = 0; m < 9; m++) {
                            if (m != j && m != k && notes[i][m][n]) {
                                verif = true;
                                notes[i][m][n] = false;
                            }
                        }
                    }
                }
                if (verif) {
                    //printf("Technique : nakedPair ligne one cell\n");
                    //printf("les cases %d et %d forme une paire nue ligne
%d\n", j + 1, k + 1, i + 1);
                    return true;
                }
            }
        }

    }
}

//printf("Grille parcourue\n");
return false;
}
```

Fichier mon_code/nakedSingle.c

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>


struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

void updateNotes(int** grid, bool*** notes, int row, int col);
float consequences_new_number(int** grid, bool*** notes,int i, int j, float*
nb_tech);

bool nakedSingle(grid_t g){
    // renvoie true si on parvient à placer un chiffre avec la technique du
naked single et place ledit chiffre
    // renvoie false sinon
    for(int i = 0; i<9; i ++){ // on parcourt la grille
        for(int j = 0; j<9; j++){
            if(g->grid[i][j]== 0){ // si la case est vide
                int count = 0 ;
                int chiffre = 0 ;
                for(int k = 0; k<9; k++){
                    if(g->notes[i][j][k]){
                        count ++;
                        chiffre = k+1 ;
                    }
                }
                if (count == 1){ // si un seul chiffre est possible
                    assert(chiffre != 0);
                    g->grid[i][j] = chiffre ;
                    //printf("Technique : nakedSingle\n");
                    //printf("row = %d, col = %d, val = %d\n", i, j, chiffre);

                    //consequences_new_number(grid, notes, i, j, nb_tech);
                    return true;
                }
            }
        }
    }
    return false ;

}

bool nakedSingle_one_cell(int** grid, bool*** notes, int i, int j, float*
nb_tech){
    if(grid[i][j]== 0){ // si la case est vide
        int count = 0 ;
        int chiffre = 0 ;
        for(int k = 0; k<9; k++){
            if(notes[i][j][k]){
```

```
                count ++;
                chiffre = k+1 ;
            }
        }
        if (count == 1){ // si un seul chiffre est possible
            assert(chiffre != 0);
            grid[i][j] = chiffre ;
            //printf("Technique : nakedSingle_one_cell\n");
            //printf("row = %d, col = %d, val = %d\n", i, j, chiffre);
            //consequences_new_number(grid, notes, i, j, nb_tech);
            return true;
        }
    }
    return false ;
}
```

Fichier mon_code/nakedTriple.c

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

/*Fonction globale naked Triple
Entrée : une grille de notes
Sortie : booléen (true si un nouveau triplet nu a été trouvé, false sinon)
Effet de bord : les notes pouvant être retirées grâce au triplet ont été
retirées
(Fonction globale appelant successivement naked triple sur les lignes,
les colonnes et les zones) */
// Ces fonctions sont inspiré du Hidden triple même si le raisonnement est
inverse
bool nakedTriple_line(bool ***notes);
bool nakedTriple_column(bool ***notes);
bool nakedTriple_zone(bool ***notes);
void free_zones(bool*** zones);

bool nakedTriple(grid_t g) {
  //printf("coucou from nakedTriple\n");
  bool ok;
  //printf("on lance sur les lignes\n");
  ok = nakedTriple_line(g->notes);
  if (!ok) {
    //printf("on lance sur les colonnes\n");
    ok = nakedTriple_column(g->notes);
  }
  if (!ok) {
    //printf("on lance sur les zones\n");
    ok = nakedTriple_zone(g->notes);
  }
  return ok;
}

bool nakedTriple_line(bool ***notes) {
  for (int i = 0; i < 9; i++) {
    int compteur[9] = {
        0, 0, 0, 0, 0,
        0, 0, 0, 0}; // tableau contenant le nombre de notes d'une case
    for (int cell = 0; cell < 9; cell++) {
      for (int j = 0; j < 9; j++) { // parcours du tableau de booléen en i,
cell
        if (notes[i][cell][j]) {    // note présente
          compteur[cell]++; // on incrémente le nombre de notes présentent dans
                            // cette case
        }
      }
    }
    for (int j = 0; j < 9; j++) {
      if (compteur[j] == 2 || compteur[j] ==3) {
        // une case ne contient que deux ou trois notes,
        //(le cas 1 note a déjà été traité par le nakedSingle,
        //mais une case avec deux notes peut faire partie d'un nakedTriple
        // sans faire partie d'un nakedPair)
        // on examine donc si elle fait partie d'un triplet nu
        for (int k = j + 1; k < 9; k++) { //on a déjà testé les cases
précédentes
          if (compteur[k] == 2 || compteur[k] ==3) { // on a trouvé une seconde
case avec seulement
                                    // deux ou trois notes
            for (int p = k + 1; p < 9; p++) {
              if (compteur[p] == 2 || compteur[p] ==3) {// on a trouvé une
troisième case avec seulement
                                      // deux ou trois notes, on teste si les
trois cases
                                      // font un triplet nu
```

```
                bool valeur[9] = {
                    false, false, false, false, false,
                    false, false, false, false}; // on utilise ce tableau pour
                                                 // enregistrer les positions
                                                 // des triplets potentiels
                int count = 0;
                for (int l = 0; l < 9; l++) {
                  if (notes[i][j][l] || notes[i][k][l] || notes[i][p][l]) {
                    valeur[l] = true;
                    count++;
                  }
                }
                // il n'y a que trois valeurs différentes dans les trois cases,
                // c'est donc un triplet
                if (count == 3) { // un triplet nu est présent, on élague si ça
                                  // n'est pas déjà fait
                  bool verif = false;
                  for (int n = 0; n < 9; n++) {
                    if (valeur[n]) {
                      for (int m = 0; m < 9; m++) {
                        if (m != j && m != k && m != p && notes[i][m][n]) {
                          verif = true;
                          notes[i][m][n] = false;
                        }
                      }
                    }
                  }
                  if (verif) {
                    //printf("Technique : nakedTriple ligne\n");
                    //printf("les cases %d, %d et %d forme un triplet nu ligne
%d\n", j + 1, k + 1, p + 1, i + 1);
                    return true;
                  }
                }
              }
            }
          }
        }
      }
    }
  }
  //printf("Grille parcourue\n");
  return false;
}
// flemme de modifier les commentaires, voir la première fonction
bool nakedTriple_column(bool ***notes) {
  for (int i = 0; i < 9; i++) {
    int compteur[9] = {
        0, 0, 0, 0, 0,
        0, 0, 0, 0}; // tableau contenant le nombre de note d'une case
    for (int cell = 0; cell < 9; cell++) {
      for (int j = 0; j < 9; j++) { // parcours du tableau de booléen en i,
cell
        if (notes[cell][i][j]) {    // note présente
          compteur[cell]++; // on incrémente le nombre de notes présentent dans
                            // cette case
        }
      }
    }
    for (int j = 0; j < 9; j++) {
      if (compteur[j] == 3 || compteur[j] == 2) { // une case ne contient que
deux notes, on examine
                                  // donc si c'est une paire nue
        for (int k = j + 1; k < 9; k++) {
          if (compteur[k] == 3 || compteur[k] == 2) { // on a trouvé une
seconde case avec seulement
                                      // deux notes, on compare alors ces notes
            for (int p = k + 1; p < 9; p++) {
              if (compteur[p] == 3 || compteur[p] == 2) {
                bool valeur[9] = {
                    false, false, false, false, false,
                    false, false, false, false}; // on utilise ce tableau pour
                                                 // enregistrer les positions
                                                 // des paires potentielles
                int count = 0;
                for (int l = 0; l < 9; l++) {
                  if (notes[j][i][l] || notes[k][i][l] || notes[p][i][l]) {
                    valeur[l] = true;
                    count++;
                  }
                }
                // il n'y a que deux valeurs différentes dans les deux cases,
                // c'est donc une paire
                if (count == 3) { // une paire nue est présente, on elague si
ça
                                  // n'est pas déjà fait
                  bool verif = false;
                  for (int n = 0; n < 9; n++) {
                    if (valeur[n]) {
                      for (int m = 0; m < 9; m++) {
                        if (m != j && m != k && m != p && notes[m][i][n]) {
                          verif = true;
                          notes[m][i][n] = false;
                        }
```

```
                    }
                  }
                }
                if (verif) {
                  //printf("Technique : nakedTriple colonne\n");
                  //printf("les cases %d, %d et %d forme un triplet nu
colonne %d\n", j + 1, k + 1, p + 1, i + 1);
                  return true;
                }
              }
            }
          }
        }
      }
    }
  }
  //printf("Grille parcourue\n");
  return false;
}

// flemme de modifier les commentaires, voir la première fonction

bool nakedTriple_zone(bool ***notes) {
  // conversion de la grille en zones
  bool ***zones = malloc(9 * sizeof(bool **));
  // on construit une grille répartis en zones :
  // plus simple pour cette étude
  // même morceau de fonction que last_remaining_cell_zone
  assert(zones != NULL);
  for (int i = 0; i < 9; i++) {
    zones[i] = malloc(9 * sizeof(bool *));
    assert(zones[i] != NULL);
    for (int j = 0; j < 9; j++) {
      zones[i][j] = notes[3*(i/3) + j/3][3*(i%3) + j%3] ;
    }
  }

  for (int i = 0; i < 9; i++) {
    int compteur[9] = {0, 0, 0, 0, 0,
                       0, 0, 0, 0};// tableau stockant le nombre d'occurence
                                   // des valeurs dans les notes (initialisé à
                                   // zéro à chaque nouvelle ligne étudié)
    for (int valeur = 0; valeur < 9; valeur++) {
      for (int j = 0; j < 9; j++) {
        if (zones[i][j][valeur]) {
          compteur[valeur]++;
        }
      }
    }
  }
  for (int i = 0; i < 9; i++) {
    int compteur[9] = {
        0, 0, 0, 0, 0,
        0, 0, 0, 0}; // tableau contenant le nombre de note d'une case
    for (int cell = 0; cell < 9; cell++) {
      for (int j = 0; j < 9; j++) { // parcours du tableau de booléen en i,
cell
        if (zones[i][cell][j]) {    // note présente
          compteur[cell]++; // on incrémente le nombre de notes présentent dans
                            // cette case
        }
      }
    }
    for (int j = 0; j < 9; j++) {
      if (compteur[j] == 3 || compteur[j] == 2) { // une case ne contient que
deux notes, on examine
                                  // donc si c'est une paire nue
        for (int k = j + 1; k < 9; k++) {
          if (compteur[k] == 3 || compteur[k]==2) { // on a trouvé une seconde
case avec seulement
                                      // deux notes, on compare alors ces notes
            for (int p = k + 1; p < 9; p++) {
              if (compteur[p] == 3 || compteur[p] == 2) {
                bool valeur[9] = {
                    false, false, false, false, false,
                    false, false, false, false}; // on utilise ce tableau pour
                                                 // enregistrer les positions
                                                 // des paires potentielles
                int count = 0;
                for (int l = 0; l < 9; l++) {
                  if (zones[i][j][l] || zones[i][k][l] || zones[i][p][l]) {
                    valeur[l] = true;
                    count++;
                  }
                }
                // il n'y a que deux valeurs différentes dans les deux cases,
                // c'est donc une paire
                if (count == 3) { // une paire nue est présente, on elague si
ça
                                  // n'est pas déjà fait
                  bool verif = false;
                  for (int n = 0; n < 9; n++) {
                    if (valeur[n]) {
```

```
                    for (int m = 0; m < 9; m++) {
                      if (m != j && m != k && m != p && zones[i][m][n]) {
                        verif = true;
                        zones[i][m][n] = false;
                        notes[3 * (i / 3) + m / 3][3 * (i % 3) + m % 3][n] =
                            false;
                      }
                    }
                  }
                }
                if (verif) {
                  //printf("Technique : nakedTriple zone\n");
                  //printf("les cases %d, %d et %d forme un triplet nu zone
%d\n", j + 1, k + 1, p + 1, i + 1);
                  free_zones(zones);
                  return true;
                }
              }
            }
          }
        }
      }
    }
  }
  //printf("Grille parcourue\n");
  free_zones(zones);
  return false;
}
```

Fichier mon_code/openness.c

```
/*
Comment coder l'ouverture d'un grille ?
- En comptant les "étapes" de recherche ?
- En réduisant le cout d'une technique proportionellement au nombre de
techniques disponibles ?




*/
```

Fichier mon_code/pointingPair.c

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct grid_s {
    int** grid ;
    bool*** notes ;
    float* nb_techniques;
};
typedef struct grid_s* grid_t ;

void updateNotes(grid_t g, int row, int col);
void free_zones(bool*** zones);

bool pointingPair(grid_t g){

    // conversion de la grille en zones
    bool ***zones = malloc(9 * sizeof(bool **));
    // on construit une grille répartis en zones :
    // plus simple pour cette étude
    // même morceau de fonction que last_remaining_cell_zone
    assert(zones != NULL);
    for (int z = 0; z < 9; z++) {
        zones[z] = malloc(9 * sizeof(bool **));
        assert(zones[z] != NULL);
        for (int c = 0; c < 9; c++) {
            zones[z][c] = g->notes[3 * (z / 3) + c / 3][3 * (z % 3) + c % 3] ;
        }
    }




    // on parcourt les zones
    for(int z = 0; z<9; z++){
        // on parcourt les valeurs
        for(int value = 0; value < 9 ; value ++){

            // on parcourt les lignes
            int count = 0 ;
            int line = -1 ;
            for(int i = 0; i<3; i++){
                // si il y a un indice dans la ligne
                if (zones[z][3*i][value]||zones[z][3*i+1][value]||zones[z]
[3*i+2][value]){
                    count ++ ;
                    line = i ;
                }
            }
            // si une seule ligne contient des indices
            if(count == 1){
                // on modifie le reste de la ligne
                assert(line!=-1);
```

```
                bool verif = false;
                for(int j=0; j<9;j++){
                    if((j<3*(z%3) || j > 3*(z%3)+2) && g->notes[3*(z/3) + line]
[j][value]){
                        g->notes[3*(z/3) + line][j][value] = false;
                        verif = true ;
                    }
                }
                // uniquement si on a modifié quelque chose !
                if(verif){
                    //printf("Technique : pointing Pair/Triple line\n");
                    //printf("Ligne = %d, zone = %d, valeur = %d\n", 3*(z/3) +
line, z, value+1);
                    free_zones(zones);
                    return true;
                }
            }



            // on parcourt les colonnes
            count = 0 ;
            int column = -1 ;
            for(int j = 0; j<3; j++){
                // si il y a un indice dans la colonne
                if (zones[z][j][value]||zones[z][j+3][value]||zones[z][j+6]
[value]){
                    count ++ ;
                    column = j ;
                }
            }
            // si une seule colonne contient des indices
            if(count == 1){
                // on modifie le reste de la colonne
                assert(column!=-1);
                bool verif = false ;
                for(int i=0; i<9;i++){
                    if((i<3*(z/3) || i > 3*(z/3)+2) && g->notes[i]
[3*(z%3)+column][value]){
                        g->notes[i][3*(z%3)+column][value] = false;
                        verif = true ;
                    }
                }
                // uniquement si on a modifié quelque chose !
                if(verif){
                    //printf("Technique : pointing Pair/Triple column\n");
                    //printf("Colonne = %d, zone = %d, valeur = %d\n",
3*(z%3)+column, z, value+1);
                    free_zones(zones);
                    return true;
                }
            }
        }
    }
    free_zones(zones);
    return false ;

}



bool pointingPair_one_zone_one_value(bool*** notes, int z, int value){

    // conversion de la grille en zones
    bool ***zones = malloc(9 * sizeof(bool **));
    // on construit une grille répartis en zones :
    // plus simple pour cette étude
    // même morceau de fonction que last_remaining_cell_zone
    assert(zones != NULL);
    for (int i = 0; i < 9; i++) {
        zones[i] = malloc(9 * sizeof(bool **));
        assert(zones[i] != NULL);
        for (int j = 0; j < 9; j++) {
            zones[i][j] = notes[3*(i/3)+j/3][3*(i%3)+j%3];
        }
    }




    // on parcourt les lignes
    int count = 0 ;
    int line = -1 ;
    for(int i = 0; i<3; i++){
        // si il y a un indice dans la ligne
        if (zones[z][3*i][value]||zones[z][3*i+1][value]||zones[z][3*i+2]
[value]){
            count ++ ;
            line = i ;
        }
    }
    // si une seule ligne contient des indices
    if(count == 1){
        // on modifie le reste de la ligne
        assert(line!=-1);
```

```
        bool verif = false;
        for(int j=0; j<9;j++){
            if((j<3*(z%3) || j > 3*(z%3)+2) && notes[3*(z/3) + line][j][value])
{
                notes[3*(z/3) + line][j][value] = false;
                verif = true ;
            }
        }
        // uniquement si on a modifié quelque chose !
        if(verif){
            //printf("Technique : pointing Pair/Triple line\n");
            //printf("Ligne = %d, zone = %d, valeur = %d\n", 3*(z/3) + line, z,
value+1);
            free_zones(zones);
            return true;
        }
    }



    // on parcourt les colonnes
    count = 0 ;
    int column = -1 ;
    for(int j = 0; j<3; j++){
        // si il y a un indice dans la colonne
        if (zones[z][j][value]||zones[z][j+3][value]||zones[z][j+6][value]){
            count ++ ;
            column = j ;
        }
    }
    // si une seule colonne contient des indices
    if(count == 1){
        // on modifie le reste de la colonne
        assert(column!=-1);
        bool verif = false ;
        for(int i=0; i<9;i++){
            if((i<3*(z/3) || i > 3*(z/3)+2) && notes[i][3*(z%3)+column][value])
{
                notes[i][3*(z%3)+column][value] = false;
                verif = true ;
            }
        }
        // uniquement si on a modifié quelque chose !
        if(verif){
            //printf("Technique : pointing Pair/Triple column\n");
            //printf("Colonne = %d, zone = %d, valeur = %d\n", 3*(z%3)+column,
z, value+1);
            free_zones(zones);
            return true;
        }
    }
    free_zones(zones);
    return false ;

}
```

Fichier mon_code/printGrid.c

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>



/*fonction print grid
Entrée : une grille
Sortie : aucune, on imprime la grille*/
void printGrid(int **grid) {
    printf("Grille actuelle : \n");
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if(grid[i][j]==0){
                printf("   ");
            }
            else{
                printf("%d  ", grid[i][j]);
            }
            if (j == 2 || j == 5) {
                printf("| ");
            }
        }
        printf("\n");
        if (i == 2 || i == 5) {
            printf("-----------------------------\n");
        }
    }
}
```