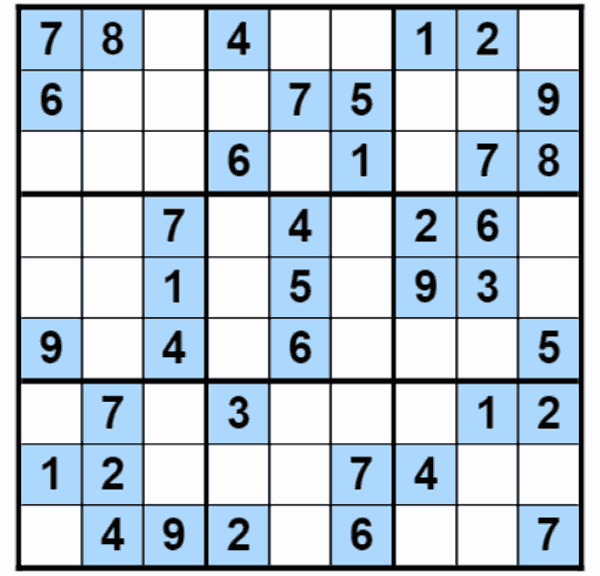


TEAM :

PANTZER DIVISION 404

Rapport M322 Le SUDOKU –TEAM PANTZER DIVISION 404



Membres de l’Equipe :

-(Chef) CERAGIOLI Arnaud ;

-(Sbire1) CUTTIVET Anthony ;

-(Sbire2) COSTA Filipe ;

-(Sbire3) BIMES Hugo ;

Table des matières : Projet SUDOKU M322

[1. Résumé 2](#_Toc469239454)

[2. Introduction 2](#_Toc469239455)

[Qu’est-ce donc que ceci ? 2](#_Toc469239456)

[Sommaire 2](#_Toc469239457)

[3. Le projet 3](#_Toc469239458)

[Présentation du problème 3](#_Toc469239459)

[1. Résolveur 3](#_Toc469239460)

[2. Générateur 3](#_Toc469239461)

[Résolution d’une grille 3](#_Toc469239462)

[1. Méthode de Backtracking 3](#_Toc469239463)

[2. Accélération du Backtracking 6](#_Toc469239464)

[3. Implémentation 8](#_Toc469239465)

[4. Résultats 8](#_Toc469239466)

[Génération d’une grille 9](#_Toc469239467)

[5. Conclusion 10](#_Toc469239468)

[6. Bibliographie 10](#_Toc469239469)

[7. Annexes 12](#_Toc469239470)

[Résolveurs 12](#_Toc469239471)

[Générateurs 16](#_Toc469239472)

# Résumé

Pour ce 3ème semestre, Le SUDOKU était le projet concerné par le module M322.

Il a dû être fait, un/plusieurs générateurs de SUDOKU ainsi que un/plusieurs résolveurs de SUDOKU mais aussi être rendu une liste de SUDOKUs générées.

Chaque résolveur avait une méthode propre de résolution et chaque générateur avait pour base ses résolveurs.

# Introduction

## Qu’est-ce donc que ceci ?

Rappelons tout d’abord ce qu’est ce jeu. Le SUDOKU est un jeu populaire apparu dans les années 1980 par Howard Garns sur un modèle du carré Latin.

Le but du jeu est de remplir la grille avec une série de chiffres tous différents qui ne se trouvent jamais plus d’une fois

- Sur une même ligne.

- Sur une même colonne.

- Dans une même sous-grille.

Les symboles sont de chiffres allant de 1 à 9 (inclus), les sous-grilles étant alors des carrées 3x3.

Au début de l’exercice, quelques valeurs sont déjà disposées dans la grille, ce qui autorise une résolution progressive du problème complet.

## Sommaire

Dans un premier temps, nous allons traiter du problème en soit suivi méthodes employées pour chacun des programmes, pourquoi ce choix a été effectué ?

Puis nous allons traiter des performances et quelles seraient les améliorations à mettre en place

On présentera alors ci-dessous, les algorithmes de résolution et de génération.

# Le projet

## Présentation du problème

### Résolveur

Le but de cet algorithme est de pouvoir résoudre n’importe quel type de SUDOKU, plusieurs méthodes pouvaient être utilisées mais le résultat devait rester le même.

On prenait alors en compte plusieurs critères pour tester la performance des algorithmes que sont : le temps, l’affichage, complexité de la méthode, le résultat sorti de la console.

### Générateur

Le but de cet algorithme est de pouvoir générer un SUDOKU, il pourra utiliser des fonctions des résolveurs, afin de mener à bien son but.

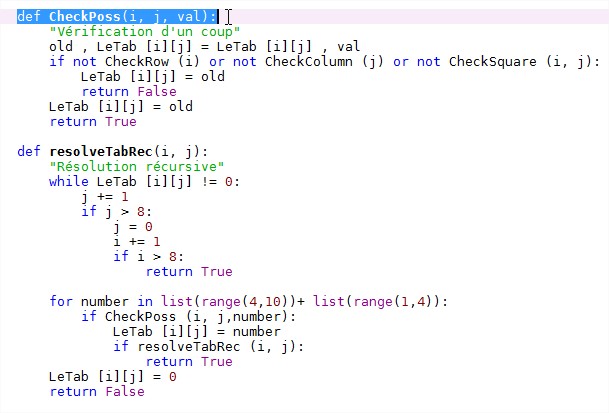
On prenait alors en compte plusieurs critères pour tester la performance de chacun des algorithmes que sont : le temps, complexité de la méthode, le résultat, la capacité à recommencer l’action si le SUDOKU généré était multi-solution.

## Résolution d’une grille

### Méthode de Backtracking

Le Backtracking (Retour sur trace), est un algorithme qui consiste à revenir en arrière sur des décisions prises afin de sortir d’un blocage (Ici affectation des valeurs sur matrice 9x9 de où les valeurs==0).

#### Résolveur 1 :

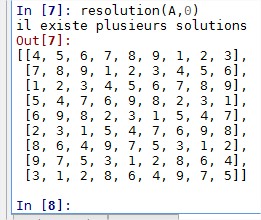


Chaque fonction Check est composé d’une liste de False ( [False]\*10 ) à laquelle on attribuera des valeurs de 1 à 9 inclus (exclusif) et seront retournées en True (si elles fonctionnent) + les conditions des règles du jeu de SUDOKU à savoir Row, Column, Square.

Il existe donc les fonctions CheckRow(i) ; CheckColumn(j) ; CheckSquare(i, j).

La fonction CheckPoss (i, j, val), va tester chacune des trois fonctions et les mettre en parallèle. Une valeur sera sorti et affecté à une cellule !=0 de la matrice 9x9.

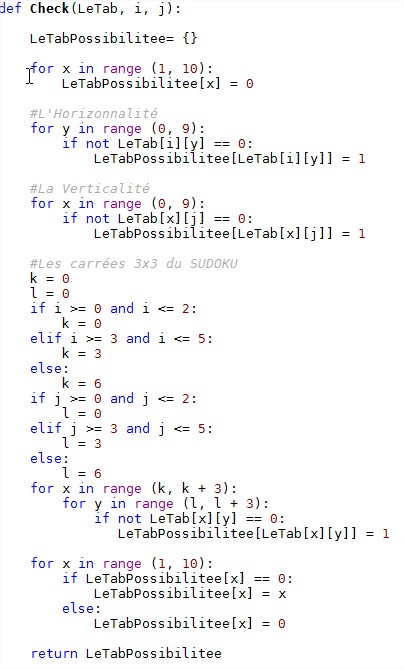
Puis on passe à la prochaine cellule en i+1 (Lorsque i arrive au bout, incrémentation de j, i=0). L’algorithme continue via resolveTabRec (i, j) une fonction récursive qui va rappeler la fonction CheckPoss(i, j , val) tant que chaque cellule de la matrice n’est pas différent de 0.

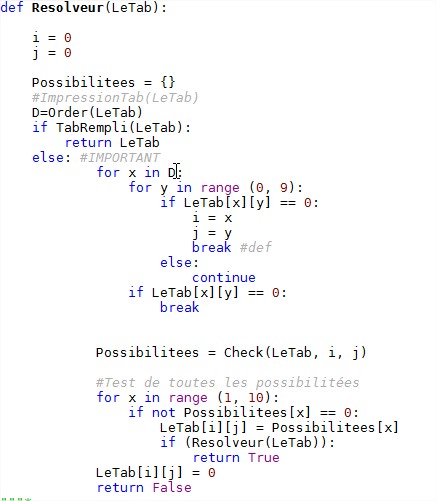
Ici, le principe de backtracking prend effet lorsque, une cellule ne peut se voir affecter une valeur car elle ne respecte pas les conditions imposées par CheckPoss. Ainsi, le système ré-effectue une attribution des valeurs en incrémentant une des solutions d’une cellule, afin de pouvoir faire continuer le programme jusqu’à ce qu’aucune autre valeur ne soit possible par rapport à chaque cellule (hormis la valeur dans chaque cellule qui fonctionne).

En cas d’erreur, aucune solution, le système retourne que la matrice entré est incorrect car elle ne respecte pas les conditions de CheckPoss.

Si la matrice entrée peut avoir plusieurs solutions possibles, en affiche une et le système répond que plusieurs réponses existent (sans toutefois les dénombrer, notamment pour le cas maximum de solutions pour une matrice composée uniquement de 0 = (9!)^9 possibilités).

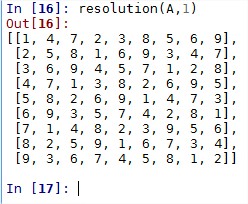
#### Résolveur 2 :





Dans la fonction Check(LeTab, i , j) , on initialise un LeTabPossibilité vide.

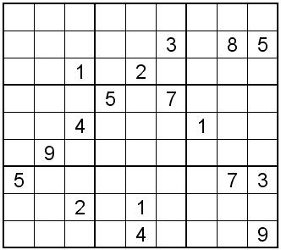
Le tableau est alors rempli pour chaque cellule par un 0. Si lors du lancement du programme, la valeur de la cellule est égale à 0, on l’incrémente et on passe dans la suivante, la valeur est alors placé dans le tableau pour une valeur x (dans un range). On aura alors un tableau comprenant chacune des valeurs possible.

La fonction Resolveur(LeTab) s’arrête à la condition que chaque cellule soit différente de 0. Sinon elle continue. Si la matrice renvoyée est composée de quelque zéros, c’est que soit le programme a été interrompu, soit aucune réponse n’est correcte.

Elle appelle la fonction Check pour attribuer à Possibilités les valeurs possibles de chaque cellule et teste donc chacune des possibilités.

Chacun de ses résolveurs avait un problème commun. Les grilles de SUDOKU, solvable difficilement par backtracking, Alors comment a t’il fallut y remédier ?

### Accélération du Backtracking

La méthode de backtracking est certes très utilisée pour résoudre des SUDOKUS numériques, toutefois il arrive qu’elle se heurte à des grilles certes solvables mais dont le temps de résolution est assez important.

Un exemple concret :

Ce sudoku comme beaucoup d’autres du même genre ont été résolus par nos 1eres versions de résolveurs en un peu moins d’une heure voir plus. Il fallait donc améliorer l’algorithme afin d’obtenir des résultats plus encourageants.

Pour cela, 2 moyens ont étés mis en place.

L’un simplet, mais qui a réduit considérablement les résultats de ~1h à 30 mn

L’autre plus complet qui a permis de faire passer juste en dessous de 2 minutes.

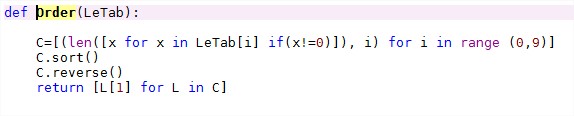
Le 1er se résume a ceci : (extrait du Solver0) 

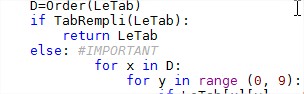
L’agencement est réordonné de façon manuelle, mais ne marche que pour quelques cas. Donc n’est pas le plus optimal selon certaines grilles. Cependant étant donné que les résultats sont là et que le temps a été réduit, on pouvait se poser la question « L’agencement de lecture joue-t-il un si grand rôle ? »

C’est pourquoi nous avons voulu tester une variante de cette 1ere solution.

Le 2ème moyen se base sur le même principe que le premier, le réagencement de la lecture de la grille de Sudoku.

Usuellement et dans nos 1er résolveurs de SUDOKU, l’ordre de lecture était de 0,0 à 9,9. (Haut gauche 🡪 bas droite). Ici, la méthode était différente et correspond à ceci : (extrait du Solver1)





La fonction Order(LeTab) va compter le nombre de valeurs différentes de 0 sur chaque ligne de la matrice 9x9, rentrer ses valeurs dans un tableau.

Ce nombre de valeurs sera affiché dans le tableau avec l’indice de la ligne correspondante puis sera trié par ordre croissant selon le nombre d’éléments pour une ligne d’indice i grâce à la fonction C.sort ().

On utilise ensuite C.reverse () pour réordonner dans l’autre sens.

Lorsque la fonction Resolveur(LeTab) va lire selon un (for x in D) au lieu d’un for x in range(0,9) classique, Le programme de résolution va résoudre en commençant d’abord par les lignes les plus remplis (reflexe qu’un humain ferait aussi) jusqu’à celle qui le sont le moins. Mais en l’occurrence, cette fois le nombre de calcul pour la machine sera très réduit, ce qui réduit aussi le temps de résolution.

### Implémentation

L’implémentation de l’algorithme s’est heurtée à plusieurs difficultés,

Il y eut :

-La réalisation des conditions du SUDOKU (L’application au code, des règles du jeu)

Particulièrement le sous-carré 3x3

-la mise en forme du backtracking

Trouver la bonne méthode

L’adapter à tout type de situations, SUDOKUs

-L’amélioration du backtracking

Trouver dans le code déjà confectionné les failles

Traiter pour chaque situation

### Résultats

## Génération d’une grille

On s’est demandé quelle pourraient être les moyens de générer une grille de SUDOKU ?

L’idée qui nous a mis sur le chemin était d’utiliser les résultats que l’on pouvait obtenir via les résolveurs.

Toutefois, on devait générer un SUDOKU à solution unique. Il était alors impossible de mettre une matrice 9x9 de 0, car son nombre de solutions est égal à 9!^9, de plus la 1re solution était toujours la même concernant la matrice 9x9 de 0.

Il a fallu donc une base pour coder cette grille de SUDOKU. Pour cela nous avons procédé comme suit :

Initialiser une matrice 9x9 de 0.

Définir la MAX\_RANGE de z, c’est-à-dire /81 combien d’élements devront être entré aléatoirement

Attribuer une valeur aléatoire à une case aléatoire (utiliser random.randint(0,8)), fonction RandomCreat (M)

Vérifier si cette valeur respecte les règles du SUDOKU, fonction checkPoint (M, a, b)

Réitérer la fonction jusqu’à arriver à MAX\_RANGE de z.

Grâce aux résolveurs précédemment crées, nous sommes capable de détecter si plusieurs solutions existent à une grille de SUDOKU donné.

Nous avons donc, exécuté un résolveur pour vérifier l’unicité de la solution au SUDOKU crée.

Si l’unicité de la solution s’avère vrai, le SUDOKU de z éléments serait retourné.

Voir plus loin :

Lorsque le résolveur a résolu le 1er SUDOKU, on pourrait afficher la grille résolu et à partir de ça, enlever un grand nombre de chiffres et déterminer si à nouveau cela correspond aux règles du SUDOKU.

Limité dans le temps et dans la réalisation + optimisations de chacun des programmes ce choix a été mis de côté.

# Conclusion

Bonne entente et compréhension du sujet par le groupe. La répartition des tâches a été bien mise en place pour permettre une meilleure avancée dans la totalité du projet.

Les problèmes rencontrées ont étés résolus avec l’aide de chacun et de M.Pantz notre enseignant encadreur.

Concernant les résultats, ils sont satisfaisants car répondent aux exigences demandées. Cependant ils pourraient être améliorés avec les apports susdits.

# Bibliographie

Cours de classes préparatoires aux grandes écoles : CERAGIOLI Arnaud (Enseignant Jean-Luc Perrin)

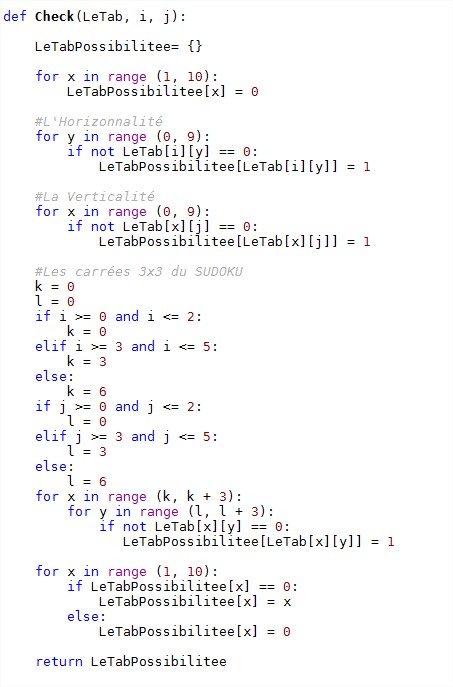
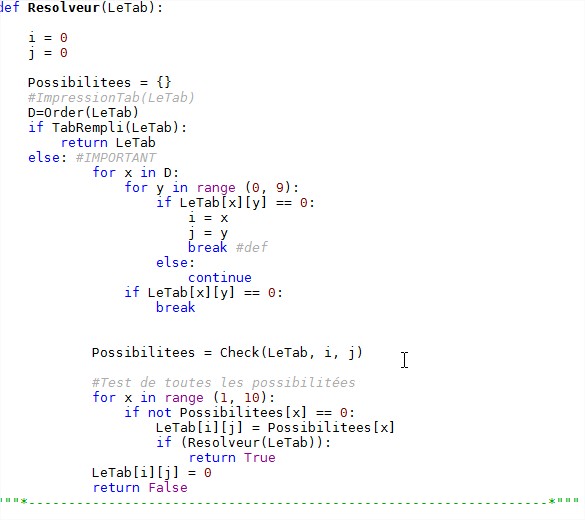
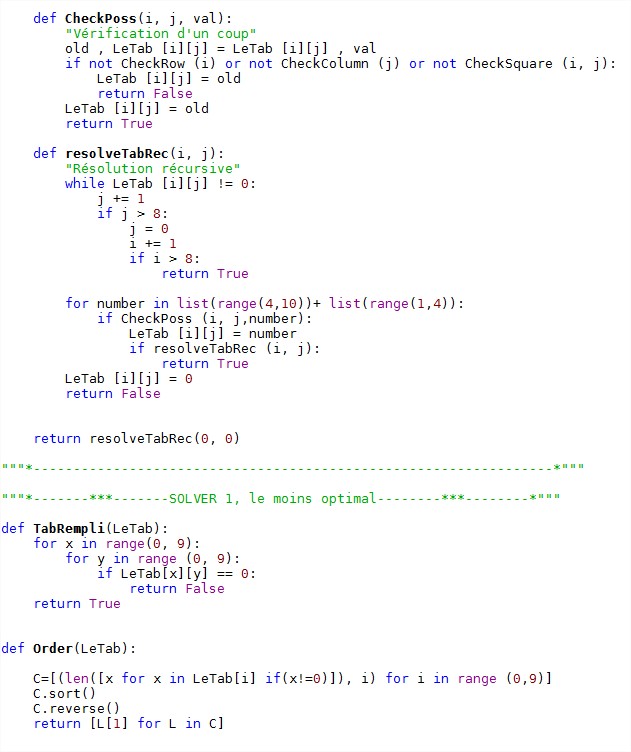
<https://fr.wikipedia.org/wiki/Math%C3%A9matiques_du_Sudoku> : Wikipédia France

Cours de module M322 , Le SUDOKU : M.Pantz

Python 201- niveau intermédiaire : Michael Driscoll

# Annexes

## Résolveurs



## Générateurs

