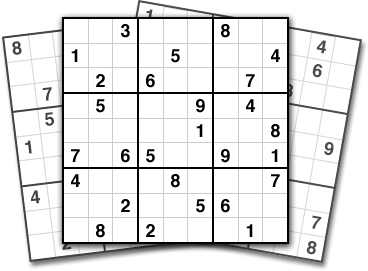
****



Projet d’INF431 – Sudoku

Mehdi KOUHEN et Timothée REBOURS

*Sujet proposé par Jean-Pierre Tillich*

Ce projet aborde deux points. Tout d’abord, la résolution d’une grille de Sudoku par *backtracking* simple dans un premier temps, et en utilisant l’algorithme X de Knuth par couverture exacte dans un second temps. Puis la génération de grilles de Sudoku de difficultés variables

Table des matières

2 Introduction 3

3 Backtracking simple 3

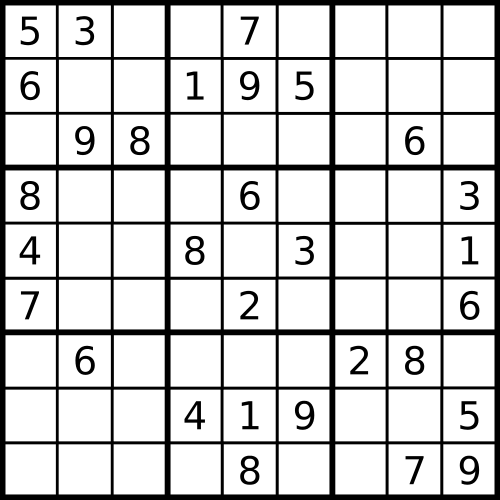
# **Introduction**

Ce projet a pour objectif de traiter de plusieurs manières la résolution d’une grille de Sudoku générique d’ordre , c’est-à-dire de taille , où est la taille d’une sous-grille et le nombre de sous-grilles par côté. Le but du sudoku est, à partir d’une grille partiellement remplie, de compléter les autres cases en respectant les contraintes suivantes :

* Chaque ligne et colonne contient exactement une fois tous les nombres de à ,
* Chaque bloc carré contient lui aussi exactement une fois tous les nombres de 1 à

Généralement, on débute d’une grille qui n’a qu’une seule solution possible. À titre d’information, on ne peut avoir une grille à solution unique si elle ne contient pas 17 entrées ou plus au départ. On appelle d’autre part grille minimale une grille ne possédant qu’une solution et qui, si on supprime n’importe quelle case, perd cette unicité.

Voici un exemple de grille minimale, et de sa solution pour  :

Pour résoudre le Sudoku, nous étudierons tout d’abord un algorithme de *backtracking* simple qui est de complexité temporelle .

Nous emploierons ensuite l’algorithme X de Knuth appliqué à la résolution de Sudoku à l’aide de listes doublement chaînées et la technique des *dancing links*.

Ensuite, nous développerons un programme permettant de générer des grilles valides de Sudoku.

Pour finir nous nous emploierons à générer des grilles minimales en combinant les deux programmes précédents.

# Backtracking simple

Cet algorithme est le plus naïf possible outre le brute-force sans vérification à chaque fois qu’une case est remplie.

Il s’agit d’un appel récursif de la méthode solve(int[][] matrix, int x, int y) qui *résout* une case donnée, où matrix est un tableau représentant le Sudoku d’ordre en cours de résolution, (x,y) sont les coordonnées de la case à remplir.

Tout d’abord on choisit les coordonnées de la case suivante en allant de gauche à droite et de haut en bas. Si on dépasse la taille cela retourne False. Si on choisit une case déjà remplie, on la *résout*.

Pour *résoudre* une case vide, on teste successivement tous les entiers entre 1 et dans cette case en appelant récursivement la méthode solve sur la case suivante pour la matrice ainsi modifiée. Si l’une d’entre elle passe tous les tests, on retourne True, sinon on remet la valeur de la case à et on retourne False.

# Résolution par un problème de couverture exacte

## Un problème de couverture exacte

Un problème de couverture exacte peut être formalisé de deux manières différentes. La première employée est formalisée par des ensembles :

On considère un ensemble , un ensemble constitué de sous-ensembles de . Le but est de trouver un sous-ensemble de contenant tous les éléments de une et une seule fois qui constituerait une *couverture exacte de* par définition. Cette formalisation est plus qu’inconfortable pour du code informatique, mais permet par la suite de mieux comprendre la conversion du problème de sudoku à un problème de couverture exacte.

La seconde manière est formalisée par des matrices binaires. Prenons un ensemble . Un sous-ensemble . La matrice binaire correspondante est :