Rapport de développement du sudoku solveur de Noé Florence

Voici le rapport de développement du projet Sudoku développé par Noé Florence (moi). Le projet est disponible à cette adresse : https://github.com/NoeFBou/Sudoku-Solver/tree/main

Ce projet a été réalisé dans le cadre du cours de génie logiciel du master 1 informatique.

Dans ce rapport, je vais détailler les étapes de développement du solveur, puis expliquer les différentes règles de déduction implémentées. Ensuite, j'aborderai les design patterns utilisés pour améliorer la conception du code. Enfin, je vais décrire et commenter en détail les classes et méthodes principales du programme

Table des matières

Objectif et contrainte du projet

L'objectif du projet est de réaliser un solveur de grilles de Sudoku en Java ou en Python qui doit essayer de résoudre une grille de Sudoku en appliquant un ensemble de règles de déduction.

Au moins trois règles de déduction devaient être définies, et chacune devait être représentée par une classe dérivant d'une même classe DeductionRule. De plus, les règles ne devaient pas utiliser d'algorithme de backtracking ou d'algorithme qui « devine » la valeur d'une case.

Pour résoudre une grille, le programme devait appliquer les règles successivement jusqu'à ce que toutes les cases soient remplies ou que le programme n'arrive plus à remplir de case. Dans ce dernier cas, l'application doit demander à l'utilisateur de mettre une valeur dans une case.

En bonus, le solveur doit attribuer un niveau de difficulté à la grille en fonction des règles utilisées pour la résoudre.

Le format d'entrée de la grille doit être un tableau de 81 éléments avec -1 quand une case est vide et les nombres sinon (on compte à partir de 1).

Étapes de développement

Le développement de ce projet s'est déroulé selon les étapes suivantes :

1. Format de la grille et parseur : J'ai commencé par établir un format pour stocker une grille dans mon programme (un tableau de chiffres) et j'ai développé le parseur qui convertit un fichier qui contient une grille.
2. Création de la classe SudokuGrid : J'ai créé une classe SudokuGrid pour gérer la grille de Sudoku. Pour faciliter le développement des règles de déduction, je l'ai enrichie avec des méthodes pour obtenir les index des lignes, colonnes, blocs, pour obtenir les « voisins » de chaque case, pour obtenir les valeurs possibles pour chaque case, etc. J'ai aussi ajouté une méthode pour afficher la grille. Toutes ces méthodes seront utilisées pour simplifier le code des règles de déduction.
3. Conception des règles de déduction :
    a. Tout d'abord, j'ai essayé de trouver des règles, de les comprendre et de les essayer sur papier à la main..
    b. Puis, j'ai choisi trois règles pas trop difficiles à implémenter que j'ai ajoutées à partir de la classe dérivable DeductionRule. Les règles choisies sont : Naked Singles (si une case n'a qu'une seule valeur possible), Hidden Singles (si une valeur dans une colonne, ligne ou bloc apparaît uniquement dans une case) et Naked Pairs (si deux cases d'une colonne, ligne ou bloc ont exactement les mêmes deux candidats).
    c. J'ai ajouté une factory pour simplifier la création des règles
4. Développement de la boucle d'application des règles : Après cela, j'ai développé la boucle qui applique les règles successivement. Tant qu'un changement est effectué par une règle, on continue à l'appliquer. Si aucun changement n'est effectué, on passe à la règle suivante.

5. Intégration de la saisie manuelle : J'ai intégré la saisie manuelle dans le cas où une grille est trop complexe.
6. Système d'évaluation de difficulté : J'ai ajouté un système d'évaluation en fonction des règles utilisées.
7. Ajout de nouvelles règles : Enfin, j'ai ajouté les règles 4 et 5 pour résoudre des grilles plus complexes. Les règles ajoutées sont Hidden Pairs (deux candidats spécifiques ne peuvent apparaître que dans deux cellules d'une même ligne, colonne ou bloc, mais ces cellules contiennent également d'autres candidats) et Pointing Pairs/Triples (lorsqu'un candidat apparaît 2 ou 3 fois dans un bloc sur une même ligne ou colonne).
8. Implémentation de design patterns : Pour finir, j'ai implémenté des design patterns pour améliorer mon code et pour respecter les contraintes de l'énoncé. J'ai ajouté un design pattern Observer en modifiant le comportement de mon programme lors de la saisie manuelle, le pattern Singleton sur la factory, le design pattern Chain of Responsibility sur l'enchaînement des règles de déduction et enfin une Façade pour améliorer la lisibilité de ma fonction main.

Commentaire sur les choix :

- Format de la grille : Pour le format de la grille, j'ai hésité à partir sur une liste de 81 éléments, un tableau double pour les lignes et les colonnes ou alors à partir sur une structure plus complexe. J'ai finalement opté pour le premier choix car je trouvais que c'était plus facile à implémenter (mais ça rend le code un peu plus difficile à lire, il faut tout le temps se demander à quelle ligne/colonne appartient chaque index).
- Exécution des règles de déduction : Pour l'exécution des règles de déduction, j'étais parti au départ sur une boucle qui exécute chaque règle une fois, mais cela rendait plus difficile l'évaluation de la difficulté. Je suis ensuite passé par une boucle qui applique les règles de déduction de manière itérative jusqu'à ce qu'aucun progrès ne soit fait. J'ai finalement fait des modifications pour implémenter le pattern Chain of Responsibility et le pattern Observer.

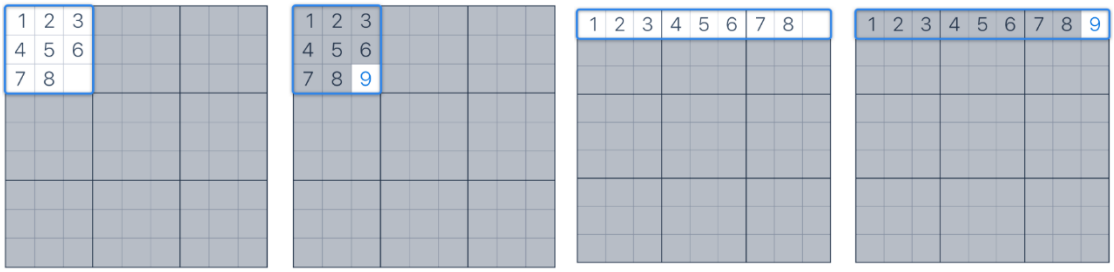
Piste d’amélioration :

- Afficher combien de fois chaque règles a été utilisé pour améliorer l’attribution de la difficulté (peut être implanter facilement via l’observer)
- Ajouter les règles x et y wings pour résoudre des grilles plus difficiles
- Ajouter de méthodes dans SudokuGrid et dans DeductionRule pour simplifier le code dans les règles de deduction.

Explication des règles de déduction utilisés :

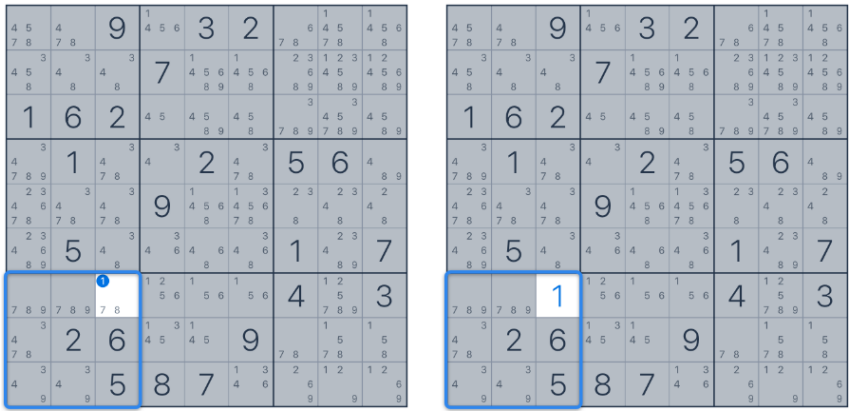
Naked Singles

Un Naked Single se produit lorsqu'une case du sudoku n'a qu'un seul candidat possible. Cela signifie que, après avoir éliminé tous les chiffres déjà présents dans la ligne, la colonne et le bloc (carré 3x3) correspondants, il ne reste qu'un seul chiffre qui peut être placé dans cette case. Exemple :



Hidden Singles

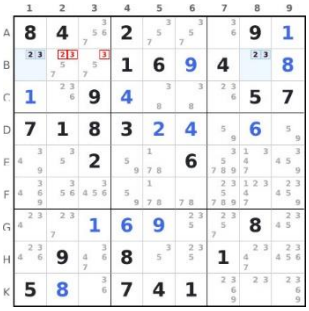
Un Hidden Single se produit lorsqu'un chiffre spécifique n'apparaît qu'une seule fois comme candidat possible dans une ligne, une colonne ou un bloc, même si la case correspondante a plusieurs candidats.



Dans le bloc en bas à gauche, le chiffre 1 ne peut être présent que dans une seule case. On peut mettre le numéro 1 cette case, puisque c'est la seule option possible.

Naked Pairs

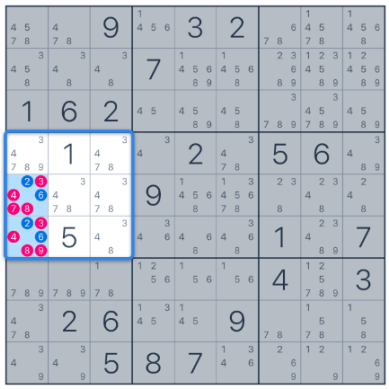
Une Naked Pair se produit lorsqu'il y a deux cases dans une même ligne, colonne ou bloc qui ont exactement les mêmes deux candidats, et uniquement ces deux candidats. Cela signifie que ces deux chiffres doivent être placés dans ces deux cases, et donc, ces candidats peuvent être éliminés des autres cases de cette unité.

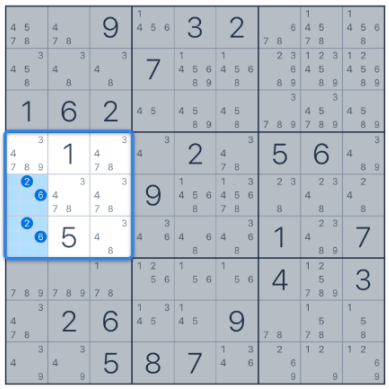


Sur la ligne B, les cellules B1 et B8 ont pour candidats les deux mêmes valeurs : 2 et 3. Ces deux valeurs peuvent donc être éliminées de la liste des candidats de toutes les autres cellules de la ligne B, ici B2 et B3.

Hidden Pairs

Une Hidden Pair se produit lorsque deux chiffres spécifiques ne sont candidats que dans les mêmes deux cases d'une ligne, colonne ou bloc, mais ces cases ont d'autres candidats supplémentaires. Les deux chiffres sont "cachés" parmi d'autres candidats dans les mêmes cases. On peut alors déduire que ces deux chiffres occupent ces deux cases et éliminer les autres candidats de ces cases.



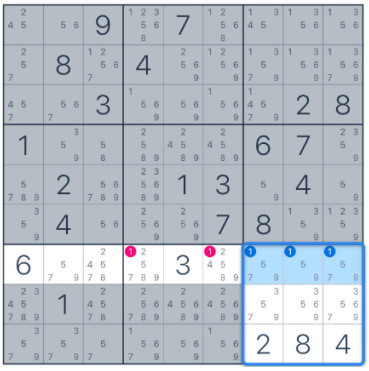


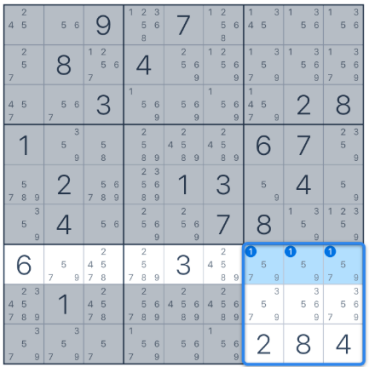
Dans cet exemple ,seules deux cases contiennent 2 et 6 dans le bloc. Cela signifie que le 2 doit occuper l'une de ces cases et le 6 l'autre.

Aucun autre nombre ne peut se trouver dans ces cases pour peut les supprimer dans candidats possibles.

Pointing Pairs/Triples (Paires/Triplets Pointants)

Une Pointing Pair ou Pointing Triple se produit lorsqu'un candidat spécifique apparaît uniquement dans une ligne ou une colonne au sein d'un bloc. Cela signifie que ce candidat doit être placé quelque part dans cette ligne ou colonne à l'intérieur du bloc, et donc, il ne peut pas être présent dans le reste de la même ligne ou colonne en dehors du bloc. On peut ainsi éliminer ce candidat des autres cases de la ligne ou colonne en dehors du bloc.





Dans ce bloc, toutes les cellules susceptibles de contenir le chiffre 1 sont situées sur une même ligne. Comme le chiffre 1 doit apparaître au moins une fois dans le bloc inférieur droit, l'une des cellules surlignées contiendra certainement le chiffre 1. Tous les autres chiffres possibles peuvent être supprimés des candidats cette rangée

Design Patterns Utilisés :

Strategy Pattern :

Implémenté par: La classe DeductionRule, DR1, DR2, DR3, DR4, DR5.

DeductionRule définit une interface commune pour les règles de déduction avec la méthode apply(). Les différentes implémentations (DR1 à DR5) représentent des stratégies spécifiques pour résoudre le Sudoku. Le SudokuSolver utilise ces stratégies de manière interchangeable pour appliquer les règles de résolution.

Chain of Responsibility Pattern:

Implémenté par: La classe DeductionRule et ses sous-classes (DR1 à DR5).

Les règles de déduction sont chaînées, chaque règle tentant d'appliquer son algorithme; si elle ne peut pas effectuer de changement, elle passe à la règle suivante.

Singleton:

Implémenté par: La classe DeductionRuleFactory.

Assure qu'une seule instance de la factory est utilisée pour créer la chaîne de règles.

Facade Pattern :

Implémenté par: La classe SudokuFacade.

Simplifie le code en encapsulant la complexité de l'initialisation de la grille et du solveur.

Factory Pattern:

Implémenté par: DeductionRuleFactory.

La classe DeductionRuleFactory est une fabrique qui instancie et retourne une liste d'objets DeductionRule.

Observer Pattern:

Implémenté par: Les classes Observable et SudokuSolver.

SudokuGrid hérite de Observable et notifie SudokuSolver des changements, ce qui permet au solveur de réagir aux modifications de la grille.

Description des classes et méthodes :

SudokuGrid.py : Classe SudokuGrid

Représente l'état d'une grille de Sudoku. Hérite de la classe Observable pour notifier les observateurs des changements de la grille.

Attributs:

- cells: Liste de 81 entiers représentant les valeurs des cellules de la grille.
- candidates: Liste de 81 ensembles représentant les candidats possibles pour chaque cellule.
- units: Liste des zones (lignes, colonnes, blocs) de la grille.
- peers: Liste des cellules associées pour chaque cellule.

DeductionRule.py : Classe DeductionRule

Classe de base abstraite pour les règles de déduction utilisées pour résoudre les puzzles Sudoku. Implémente le design pattern Chaîne de responsabilité.

Méthodes:

- set_next(self, rule): Définit la prochaine règle dans la chaîne.
- apply(self, grid): Méthode abstraite à implémenter par les sous-classes
- handle(self, grid): Tente d'appliquer la règle; si aucun changement n'est effectué, passe à la règle suivante.

DR1.py : Classe DR1

Implémente la règle de déduction Naked. Pour chaque cellule, si elle est vide et n'a qu'un seul candidat, assigne ce candidat à la cellule.

DR2.py : Classe DR2

Implémente la règle de déduction Hidden. Pour chaque unité (ligne, colonne, bloc), si un candidat n'apparaît qu'une seule fois parmi les candidats des cellules vides, assigne ce candidat à cette cellule.

Étape 1: Parcourt chaque unité de la grille.

Étape 2: Compte le nombre d'occurrences de chaque candidat dans les cellules de l'unité.

Étape 3: Si un candidat n'apparaît qu'une seule fois et que la cellule correspondante est vide, il assigne ce candidat à cette cellule.

DR3.py : Classe DR3

Implémente la règle de déduction Naked Pairs. Pour chaque colonne, ligne, bloc, identifie les paires de cellules ayant exactement les mêmes deux candidats et élimine ces candidats des autres cellules de la zone.

Étape 1: Parcourt chaque zone (colonne, ligne, bloc) de la grille.

Étape 2: Recherche les cellules avec exactement deux candidats identiques.

Étape 3: Si une telle paire est trouvée, élimine ces deux candidats des autres cellules de l'unité.

DR4.py : Classe DR4

Implémente la règle de déduction Hidden Pairs. Dans une unité, si deux candidats n'apparaissent que dans exactement deux cellules, élimine les autres candidats de ces cellules.

Étape 1: On parcourt de chaque zone de la grille.

Étape 2: On créé une map des candidats aux cellules où ils apparaissent. Ce mapping permet de savoir rapidement dans quelles cellules de la zone chaque candidat est présent.

Étape 3: On identification des paires de candidats qui n'apparaissent que dans les mêmes deux cellules. On calcule cells_c1 et cells_c2, les ensembles des indices de cellules où chaque candidat apparaît, on trouve les common_cells, les cellules communes aux deux candidats et on vérifie que les deux candidats n'apparaissent que dans ces deux cellules

Étape 4: Dans ces cellules, on élimine tous les autres candidats sauf les deux pairs cachés.

DR5.py : Classe DR5

Implémente la règle de déduction Pointing Pairs/Triples. Si dans un bloc, tous les candidats d'un nombre sont confinés à une seule ligne ou colonne, élimine ce candidat des autres cellules de cette ligne ou colonne en dehors du bloc.

Étape 1: On parcourt chaque bloc de la grille et on crée un dictionnaire candidate_cells qui mappe chaque candidat aux cellules du bloc où il apparaît.

Étape 2: Pour chaque candidat, on identifie les cellules dans lesquelles il apparaît. On récupère les lignes et les colonnes où il apparaît dans le bloc.

Étape 3: On vérifie si ces cellules sont toutes sur la même ligne ou colonne.

Étape 4: Si oui, on élimine ce candidat des autres cellules de la ligne ou colonne correspondante en dehors du bloc

DeductionRuleFactory.py : Classe DeductionRuleFactory

Implémente le design pattern Singleton pour créer une factory qui crée une chaîne de règles de déduction. S'assure qu'une seule instance de la classe est créée

SudokuSolver.py : Classe SudokuSolver

Applique les règles de déduction pour résoudre une grille de Sudoku.

Méthodes:

- __init__(self, grid): Initialise le solveur avec une grille donnée.
- update(self, observable, event=None): Méthode appelée lorsque la grille change (implémentation de l'observateur).
- apply_rules(self): Applique les règles de déduction de manière itérative jusqu'à ce qu'aucun progrès ne soit fait. La vérification de l'incohérence sert à détecter les situations où la grille est impossible à résoudre avec les informations actuelles.
- solve(self): Tente de résoudre la grille.
- user_input(self): Demande à l'utilisateur d'entrer une valeur manuellement si nécessaire.
- evaluate_difficulty(self): Évalue le niveau de difficulté du puzzle en fonction des règles utilisées.

SudokuFacade.py : Classe SudokuFacade

Sert de façade pour simplifier l'utilisation du solveur de Sudoku.

Attributs:

- grid: Instance de SudokuGrid.
- solver: Instance de SudokuSolver.

Méthodes:

- __init__(self, initial_values): Initialise la grille et le solveur avec les valeurs initiales.
- solve(self): Résout le puzzle et affiche la solution.

Observable.py : Classe Observable

Cette classe implémente le design pattern Observateur. Elle est utilisé pour savoir si un changement à été fait sur la grille et permet au solveur d'être informé des changements dans la grille et de réagir en appliquant les règles de déduction pour avancer dans la résolution de la grille.

Main.py

Point d'entrée du programme. Gère la lecture du fichier d'entrée, l'initialisation du solveur et l'affichage du résultat.