Rapport Sudoku

Cherigui Allah-Eddine & Younes Mohamed

Introduction :
Ce projet vise à développer un solveur de Sudoku en utilisant des règles de déduction pour remplir progressivement les cellules d'une grille. Le programme suit des principes fondamentaux d'ingénierie logicielle, appliquant au moins trois règles de déduction afin de résoudre les Sudokus, ou au besoin, demander à l'utilisateur d'intervenir. Le programme a été développé en **Java** et utilise des design patterns pour structurer et organiser le code de manière efficace et maintenable.

Problématiques :

1. **Structurer et appliquer des règles de déduction en séquence** : Chaque règle (DR1, DR2, DR3) doit être exécutée pour résoudre autant de cellules que possible avant de passer à la règle suivante.
2. **Gérer les candidats possibles pour chaque cellule** : Pour les cellules encore indéterminées, il est nécessaire de suivre les chiffres candidats, soit ceux qui peuvent potentiellement remplir ces cellules sans violer les règles de Sudoku. Cela complique la gestion des données et nécessite une structure pour les manipuler efficacement.
3. **Prévoir un retour en arrière en cas d'incohérence** : Lorsque le solveur est dans une impasse, il doit pouvoir revenir à un état antérieur et proposer une autre valeur sans recommencer l'ensemble du processus depuis le début.
4. **Évaluation des candidats et déductions complexes** : Les règles DR2 et DR3 impliquent des déductions complexes, notamment la gestion de paires et de triplets cachés dans les lignes, colonnes et blocs, rendant indispensable l'utilisation de structures de données comme des ensembles (`HashSet`).

Description Générale du Programme :

Le solveur prend en entrée une grille de Sudoku vide et procède en ajoutant les valeurs de départ une par une. Les règles de déduction sont appliquées chaque fois qu'une cellule reçoit une valeur. Si, après l'application des règles, la grille n'est toujours pas complète, le solveur demande à l'utilisateur de fournir une valeur pour une cellule. En cas d'incohérence détectée dans la grille, un message invitant à redémarrer le processus est affiché.

Format d'Entrée

L'entrée est un fichier texte contenant une ligne pour chaque ligne de la grille, les valeurs étant séparées par des virgules. Les cellules vides sont représentées par des zéros. Exemple de fichier d'entrée :

Copier le code

```
5,3,0,0,7,0,0,0,0

6,0,0,1,9,5,0,0,0

0,9,8,0,0,0,0,6,0

8,0,0,0,6,0,0,0,3

4,0,0,8,0,3,0,0,1

7,0,0,0,2,0,0,0,6

0,6,0,0,0,0,2,8,0

0,0,0,4,1,9,0,0,5
```

```
0,0,0,0,8,0,0,7,9
```

Structure du Programme et Design Patterns

Pour organiser le code et faciliter la maintenance, quatre design patterns principaux ont été utilisés : **Factory**, **Strategy**, **Memento** et des structures de données spécifiques pour gérer les candidats(Coord, Int_CoordHashSet_Tuplet, Int_CoordHashSet_Tuplet).

.

3.1 Pattern Factory pour les Règles de Déduction

Objectif : Instancier dynamiquement les règles de déduction (`DR1`, `DR2`, `DR3`) tout en isolant leur création de la logique principale du solveur. Cela rend le code extensible en facilitant l'ajout ou la modification de nouvelles règles.

Implémentation : Une `List_DR` crée les instances des règles en fonction du niveau de difficulté. Le solveur principal n'a ainsi qu'à demander à la factory de fournir la règle appropriée sans connaître les détails de leur instanciation.

3.2 Pattern Strategy pour Appliquer les Règles en Séquence

Objectif : Appliquer les règles de déduction dans un ordre prédéfini pour maximiser la résolution des cellules sans intervention de l'utilisateur.

Implémentation : Le solveur implémente l'interface `DR_strat`, qui inclut les différentes règles comme stratégies. Ainsi, les règles DR1, DR2, et DR3 peuvent être appliquées l'une après l'autre sans changer la structure du solveur. Cela permet une flexibilité d'application et une adaptation du solveur aux différents niveaux de complexité des grilles.

3.3 Pattern Memento pour le Retour en Arrière

Objectif : Permettre au solveur de sauvegarder et restaurer des états précédents de la grille afin de pouvoir annuler des changements si une incohérence est détectée.

Implémentation : Le memento enregistre l'état complet de la grille avant chaque modification. Si une incohérence est détectée, le solveur peut restaurer la grille dans l'état antérieur, évitant ainsi de réinitialiser entièrement le processus. Cette fonctionnalité garantit également une meilleure expérience utilisateur en réduisant le nombre de redémarrages nécessaires.

3.4 Structures de Données pour Gérer les Candidats

Objectif : Simplifier la gestion des candidats possibles pour chaque cellule vide en fonction des règles de Sudoku, qui exigent une vérification des lignes, colonnes et blocs.

Implémentation :

- `Coord` : Cette classe représente les coordonnées d'une cellule spécifique dans la grille et permet de comparer efficacement des positions de cellules.
- `Int_CoordHashSet_Tuplet` et `Int_IntHashSet_Tuplet` : Ces classes facilitent la gestion des candidats pour chaque cellule. `Int_CoordHashSet_Tuplet` gère des ensembles de coordonnées pour les candidats potentiels, et `Int_IntHashSet_Tuplet` suit les chiffres candidats pour chaque cellule indéterminée.

Les règles de déduction DR2 et DR3 utilisent ces classes pour détecter des paires ou triplets cachés, en analysant les candidats dans chaque ligne, colonne et bloc. Par exemple, DR2 vérifie les triplets cachés en ligne et colonne et ajuste les candidats de certaines cellules en fonction de l'analyse des occurrences de chaque chiffre candidat.

4. Explication des Règles de Déduction

4.1 Règle DR1 : Remplissage Direct (balayage)

Cette règle identifie les cellules avec un seul candidat possible et assigne immédiatement ce candidat à la cellule. DR1 est utilisée pour résoudre les grilles les plus simples.

4.2 Règle DR2 : Identification des Triplets Cachés

Comment ça marche : DR2 détecte les Triplets cachées dans les lignes, colonnes ou blocs. Si deux cellules contiennent exactement les mêmes deux candidats, il est possible de restreindre ces candidats à ces cellules uniquement, ce qui réduit les options pour les autres cellules dans la même région.

Structure et Étapes Clés :

Initialisation de `newGrille`

```
Int_IntHashSet_Tuplet[][] newGrille = new
Int_IntHashSet_Tuplet[9][9];
```

`newGrille` est une représentation de la grille initiale, où chaque cellule est soit un `Int_IntHashSet_Tuplet` (valeur fixe) ou un ensemble de candidats possibles si la cellule est vide (`-1` dans `grille`). Cette nouvelle grille est utilisée pour stocker les valeurs potentielles.

- Pour chaque cellule vide, la méthode `trouverPossibles` est appelée pour générer les candidats possibles en fonction des règles du Sudoku (par exemple, en excluant les chiffres déjà présents dans la même ligne, colonne ou bloc).

Recherche de "Triplets Cachés" dans les Lignes et Colonnes

Pour chaque ligne et colonne, le code cherche des triplets de cellules partageant les mêmes candidats potentiels.

1. **Initialisation des Comptes d'Occurrences**
   Deux tableaux `occurences_ligne` et `occurences_colonne` de type `Int_CoordHashSet_Tuplet` sont créés pour stocker les occurrences des candidats dans chaque cellule de la ligne ou colonne.
2. **Mise à Jour des Comptes d'Occurrences**
   Pour chaque cellule vide de `newGrille`, on parcourt l'ensemble des candidats. Si un candidat particulier est trouvé dans une cellule, il est ajouté à l'ensemble `coordhashset` associé dans `occurences_ligne` ou `occurences_colonne`. Ce processus

permet de recenser combien de fois chaque chiffre apparaît dans une ligne ou colonne.

3. **Détection des Triplets Cachés**
   Si un candidat a une occurrence de trois dans une ligne ou colonne, le code vérifie si trois cellules partagent le même ensemble de candidats, en utilisant `equals` pour comparer les `coordhashset`.
   Lorsqu'un triplet caché est détecté, les candidats des trois cellules sont restreints aux valeurs du triplet trouvé, ce qui réduit les autres possibilités.

Gestion des Blocs 3x3

Le processus pour chaque bloc de 3x3 est similaire à celui des lignes et colonnes. La différence est que le code calcule les coordonnées de chaque bloc 3x3 (en fonction de `blocLigne` et `blocColonne`) et effectue les mêmes étapes pour détecter et traiter les triplets cachés.

Réduction de la Grille avec `boucle_interieur`

Après chaque cycle principal (représenté par `boucle_exterieur`), le code effectue une boucle interne `boucle_interieur`, qui cherche à remplir les cellules dès qu'elles n'ont qu'un seul candidat possible.

- Pour chaque cellule de `newGrille`, si un seul candidat reste, la cellule de **`grille`** est mise à jour avec cette valeur unique, ce qui résout cette cellule et permet de propager les réductions possibles.

4.3 Règle DR3 : Identification des X-Wings

La règle DR3 repose sur la stratégie des *X-Wings*, une technique avancée en résolution de Sudoku. Cette méthode consiste à identifier deux lignes ou deux colonnes contenant chacune exactement deux occurrences possibles d'un candidat dans les mêmes positions. Si cette disposition forme un motif en "X" entre les lignes et les colonnes, alors ce candidat peut être éliminé des autres cellules de ces lignes et colonnes.

L'application de la technique des X-Wings est particulièrement utile dans les grilles de difficulté élevée où les candidats restants ne peuvent pas être déterminés par des techniques de base ou des paires cachées. Grâce à cette

approche, le solveur élimine efficacement des candidats et réduit les possibilités, ce qui facilite la progression vers une solution complète.

Méthode `liste_possible(int[][] grille)`

- Cette méthode génère une liste des candidats possibles pour chaque cellule vide dans une grille.
- Elle utilise une boucle double (`i` et `j` pour chaque ligne et colonne) pour parcourir chaque cellule.
- Si une cellule est vide (**`grille[i][j] == -1`**), elle appelle **`trouverPossibles(i, j, grille)`** pour obtenir un `HashSet` des valeurs possibles pour cette cellule.
- **Résultat** : Elle retourne une **`ArrayList<HashSet<Integer>>`** contenant les candidats possibles pour chaque cellule de la grille.

Méthode `catch_xwing(ArrayList<HashSet<Integer>> candidats)`

- Cette méthode implémente la stratégie X-Wing :
  - **But** : Identifier les cas où un candidat (chiffre de 1 à 9) apparaît exactement deux fois dans deux lignes ou deux colonnes spécifiques, créant une configuration en « X » qui permet de restreindre les valeurs dans les autres cellules de ces lignes/colonnes.
  - Elle commence par parcourir chaque chiffre (`num`) de 1 à 9 pour vérifier si ce candidat forme un X-Wing.
  - **Détail du traitement** :
    1. Elle construit deux listes, `lignesCandidats` et `colonnesCandidats`, pour stocker les indices de lignes et colonnes où le candidat apparaît.
    2. Ensuite, elle parcourt chaque cellule de la grille pour chaque candidat possible dans `candidats`. Si un candidat contient `num`, elle ajoute l'indice de la ligne dans `lignesCandidats` et celui de la colonne dans `colonnesCandidats`.
    3. Elle compare chaque paire de lignes ou de colonnes pour voir si elles partagent les mêmes colonnes ou lignes pour ce candidat. Si c'est le cas, cela forme un X-Wing.
    4. Lorsque le X-Wing est détecté, la méthode élimine ce candidat des autres cellules dans ces lignes ou colonnes.

- **Résultat** : Retourne une liste `modifications` des indices de cellules où des candidats ont été éliminés, ce qui facilitera la mise à jour de la grille.

Méthode `applyDR(int[][] grille)`

- **But** : Appliquer la méthode X-Wing de manière répétitive jusqu'à ce que la grille ne change plus ou qu'une limite d'itération (`maxIterations`) soit atteinte.
- **Détails du processus** :
    - Utilise une boucle `do-while` avec une condition de `modifications` pour exécuter les stratégies jusqu'à ce qu'il n'y ait plus de changements.
    - À chaque itération :
        - Elle génère la liste des candidats avec `liste_possible(grille)`.
        - Elle applique `catch_xwing(listeCandidats)` pour détecter les motifs X-Wing et obtenir les modifications.
        - Si des modifications sont faites, elle met à jour la grille avec `mettreAJourListe`.
        - Elle applique aussi une stratégie de placement unique (`appliquerPlacementsUniques`) qui remplit les cellules avec un candidat unique possible.
- **Résultat** : La méthode modifie directement la grille fournie pour la rapprocher de la solution.

Méthode `mettreAJourListe(ArrayList<Integer> modifs, ArrayList<HashSet<Integer>> liste, int[][] grille)`

- **But** : Mettre à jour les candidats des cellules modifiées.
- **Détails** :
    - Pour chaque cellule modifiée dans `modifs`, elle appelle `trouverPossibles(row, col, grille)` pour recalculer les candidats possibles.
    - **Exemple** : Si le candidat 3 est éliminé de la cellule en (2, 5), `trouverPossibles` est appelé pour ajuster les candidats restants.

Méthode
`appliquerPlacementsUniques(ArrayList<HashSet<Integer>> liste, int[][] grille)`

- **But** : Vérifier chaque cellule pour placer les candidats uniques.
- **Détails** :
    - Elle parcourt chaque cellule dans la grille. Si une cellule a un seul candidat possible (`size == 1`), ce candidat est directement placé dans la grille.
-

5. Résolution de Grilles de Difficultés Différentes

La résolution de grilles de différents niveaux de difficulté repose sur les règles de déduction :

- **Facile** : La grille est résolue entièrement avec DR1.
- **Moyenne** : DR2 complète les étapes que DR1 n'a pas pu résoudre.
- **Difficile** : La grille nécessite DR3 pour être résolue.
- **Très difficile** : DR3 ne suffit pas, alors nous pouvons entre nous-même un chiffre dans une case et le programme va vérifier si c'est cohérent.

Conclusion

Ce projet de solveur de Sudoku démontre l'utilité de design patterns tels que Factory, Strategy, et Memento pour créer une architecture robuste et flexible. Les structures de données utilisées facilitent la gestion des candidats possibles, tandis que le pattern Memento permet de gérer les incohérences efficacement. Grâce à cette approche modulaire, le solveur peut être facilement étendu pour gérer des règles plus complexes ou intégrer d'autres méthodes de résolution avancées.