Rapport de projet- Développement du jeu de kakuro

1. Introduction

Ce projet avait pour objectif de modéliser le célèbre jeu de Kakuro et d'implémenter un *solver* permettant de vérifier, pour une grille donnée, s'il existe une solution valide.

Le Kakuro est un jeu de logique dans lequel le joueur doit remplir une grille avec des chiffres de 1 à 9, en respectant deux types de contraintes :

- Chaque groupe de cellules (en ligne ou en colonne) doit totaliser une somme précise indiquée dans une cellule de référence.

- Aucun chiffre ne peut être répété dans un même groupe (ligne ou colonne associée à une somme).

Les sommes à atteindre varient généralement entre 3 et 45 (la somme maximale possible avec des chiffres uniques de 1 à 9).

Ce projet a été réalisé en binôme, en suivant les consignes fournies dans le sujet. Nous avons mis en œuvre une architecture orientée objet pour représenter les différentes composantes du jeu, ainsi que deux algorithmes de résolution automatique.

2. Choix de modélisation

Nous avons accordé une attention particulière à la modélisation orientée objet, dans le but de rendre notre architecture cohérente, maintenable et extensible. Le projet repose sur une séparation claire des responsabilités entre les différentes classes. **Le diagramme complet se trouve dans UML.png.**

a. Représentation des cellules de la grille:

Nous avons modélisé les différentes cellules du Kakuro à l'aide d'un ensemble de classes spécifiques, dérivées d'une interface commune **Cells**. Cela nous permet de gérer uniformément tous les types de cellules tout en personnalisant leur comportement.

- **Cell_vide** : cellule à remplir par le joueur ou le solveur, avec des contraintes de somme et d'unicité.

- **Cell_sum** : cellule contenant une contrainte de somme horizontale et/ou verticale.
- **Cell_noir** : cellule noire servant uniquement de séparation visuelle dans la grille.

Cette approche nous a permis de manipuler les cellules de manière polymorphe, ce qui a simplifié le traitement de la grille.

b. Lecture et création des cellules : application du *Factory Pattern*

Une partie essentielle de notre architecture repose sur la classe **Factory**, qui centralise le processus de construction de la grille depuis un fichier d'entrée. Cette classe s'appuie sur deux interfaces complémentaires :

- **Reader** : responsable de la lecture du fichier (texte ou JSON),
- **ICellFactory** : responsable de la création des objets ***Cells*** à partir de tokens ou données lues.

Ce découplage nous a permis d'appliquer le **design pattern Factory** de manière modulaire.

- **TextReader** et **JsonReader** lisent respectivement des fichiers texte et JSON.
- **TextCellFactory** et **JsonCellFactory** créent des cellules spécifiques à chaque format.

La classe **Factory** combine dynamiquement ces deux composants pour construire une Grille complète via la méthode **buildGrid()**.

c. Classe Grille : structure et gestion de la grille

La classe **Grille** représente la structure logique du plateau de Kakuro. Elle encapsule une matrice de cellules Cells[][] et fournit des méthodes pour interagir avec celles-ci. Ses responsabilités incluent :

- **Stockage de la grille** sous forme de tableau 2D (cellules), avec les dimensions (**rows, cols**) adaptées au fichier lu.
- **Accès/modification des cellules** via les méthodes **setCell(i, j, cell)** et **getCell(i, j)**, permettant donc au joueur de jouer.
- **Affichage** de la grille dans une représentation lisible par l'utilisateur, notamment pour le mode console.

La construction de la grille est entièrement déléguée à la classe Factory. **Grille** reste donc indépendante du format d'entrée et ne s'occupe que de la logique liée aux cellules elles-mêmes.

d. Classe Solver : abstraction de la résolution

La classe **Solver** est une classe **abstraite**, qui définit une interface unique : **solution(grille)**. Cette méthode prend une instance de ***Grille*** et retourne une grille résolue (ou modifie l'originale selon l'implémentation).

Deux classes concrètes héritent de solver **: BruteForceSolver et FilteredSolver**

Grâce à cette abstraction, il est possible d'ajouter de nouveaux solveurs à l'avenir.

e. Classe Game : gestion globale du jeu

La classe **Game** est le point d'entrée principal du programme. Elle centralise le flux d'exécution du jeu Kakuro, depuis le chargement d'une grille jusqu'à sa résolution. Elle encapsule une instance de ***Grille*** ainsi qu'un objet ***Solver***, et fournit une interface simple pour effectuer les actions principales du jeu.

La classe Game constitue donc une couche de coordination qui lie ensemble les composants spécialisés du projet (lecture, modélisation, résolution), tout en maintenant un bon niveau d'abstraction.

3. Explication synthétique des algorithmes :

Dans notre implémentation on a utilise deux algorithmes pour le solver qui sont :

Brute Force Solver : cet algo utilise deux fonctions, solution et isvalid pour résoudre la grille.
solution(Grille& g) :

1. Parcourt la grille ligne par ligne, colonne par colonne.
2. Dès qu'une cellule vide (*Cell_vide*) avec valeur 0 est trouvée :
    - Essayez chaque valeur possible de 1 à 9.
    - Attribuer temporairement la valeur à la cellule.
    - Appelle récursivement solution(g) pour poursuivre la tentative.
    - Si une solution est trouvée, elle est immédiatement retournée.
    - Sinon, l'algorithme annule la tentative en remettant 0.

3. Si toutes les cellules sont remplies, un dernier passage valide toute la grille avec la fonction *isValid()*

isValid(Grille& g, int row, int col) :
Cette fonction vérifie que la cellule (row, col) respecte les contraintes horizontales et verticales :

- Elle remonte vers la gauche ou le haut pour trouver la cellule de type *Cell_sum* associée.
- Elle avance ensuite dans la ligne ou la colonne pour :
    - Accumuler les valeurs déjà remplies,
    - Compter les cellules vides restantes,
    - S'assurer qu'il n'y a aucun doublon

    - Vérifier que la somme actuelle ne dépasse pas la somme cible.

Filtered Solver : L'algorithme implémenté dans FilteredSolver repose sur une combinaison de backtracking classique et de filtrage de contraintes locales pour optimiser les performances.

Le FilteredSolver repose sur une stratégie de backtracking optimisée par filtrage de contraintes. L'algorithme démarre avec la méthode solution(), qui réinitialise les cellules vides et appelle récursivement solveKakuro(). Cette fonction explore la grille cellule par cellule : pour chaque Cell_vide, elle tente les chiffres de 1 à 9. Avant d'assigner une valeur, elle appelle isValid(), qui centralise la logique de vérification. Cette dernière invoque à son tour deux fonctions spécialisées : checkHorizontal() et checkVertical(), qui examinent respectivement la ligne et la colonne contenant la cellule. Ces fonctions identifient la séquence complète de cellules liées à une somme, vérifient l'unicité des valeurs, et comparent la somme partielle à la somme cible extraite d'une Cell_sum. Si la

valeur est valide, SolveKakuro() continue récursivement. Sinon, elle annule la tentative (backtracking). Cette architecture en couches permet une séparation claire des responsabilités et une propagation efficace des contraintes à chaque étape de la résolution.

4. Fonctionnalités unique de l'application :

En complément de la résolution automatique, notre application propose un mode interactif de jeu, enrichi par trois niveaux de difficulté : Facile, Difficile et Défi. Ces modes modifient les règles de la partie et sont définis dans la classe Game.

a. Mode Facile

Le joueur dispose d'un temps illimité et d'un nombre illimité de tentatives. Il peut remplir la grille à son rythme. Ce mode est destiné à l'apprentissage ou à une résolution sans pression.

b. Mode Difficile

Le temps est limité à 120 secondes et le joueur dispose d'un nombre restreint de tentatives (10). Chaque tentative consiste à entrer une valeur dans une cellule. Lorsqu'il n'a plus de tentatives ou que le temps est écoulé, la partie s'arrête.

c. Mode Défi

C'est une version plus immersive : le temps est chronométré (par exemple 180 secondes en "moyen" ou 120 en "difficile"), les erreurs sont limitées, et une vérification finale automatique est effectuée à la fin. Si la grille est correctement remplie, un message de victoire s'affiche. Sinon, le programme propose une solution générée automatiquement via le FilteredSolver, renforçant l'aspect pédagogique.

Répartition du travail entre étudiants

Le projet a été réalisé en binôme par Mamadou Tanou Diallo et Djiby Kebe. La plus grande partie a été développée de manière collaborative, tandis que d'autres ont été réparties pour optimiser le travail.

Travail commun :

- Conception de l'architecture générale du projet (diagrammes de classes, modélisation orientée objet).
- Définition des types de cellules (Cell_vide, Cell_sum, Cell_noir)
- Implémentation des classes Grille et game.
- Mise en place des solveurs et tests fonctionnels sur les grilles.
- Ajout des différents modes de difficulté et du mode interactif dans la classe Game.
- Intégration des modes

Individuellement :

- **Mamadou Tanou Diallo** : prise en charge principale des fonctions de la classe Factory et ses sous classes et de la fonction FilteredSolver.
- **Djiby Kebe** : développement du BruteForceSolver, des fonctions de validation et des cellules.