LEHNERT William

Projet Sudoku (TP3)

Présentation de la Classe Sudoku

À la suite de la lecture entière du sujet, j'ai immédiatement commencé à écrire cette Classe en sachant qu'on travaillera sur des Sudokus de taille variable. J'ai donc de suite créé cette classe et ses méthodes dans le but de l'adapter à un Sudoku de n'importe quelle taille.

Attributs		
Type	**Nom**	**Explication**
Vector<vector<int>>	***Board***	Tableau bidimensionnel représentant a grille de **Sudoku.**
int	***complexity***	Niveau de difficulté du **Sudoku.**
Int	***size***	Taille des lignes et des colonnes du **Sudoku.**

Les deux premiers constructeurs de la classe créent une grille classique de 9x9. Le dernier prend en compte la taille de la grille en plus. On a aussi des Getters/Setters simples et une surcharge de l'opérateur << pour afficher le **Sudoku**. Pour représenter la grille de **Sudoku**, j'ai choisi d'utiliser un tableau bidimensionnel à l'aide de vecteurs car j'ai appris à les utiliser dans ce but.

Methods		
Retour type	**Nom**	**Explication**
Int	***getCase(…)***	Permet d'obtenir la valeur d'une case.
Void	***clearBoard(…)***	Nettoie toute la grille (remplie de 0).
Void	***randomFilling(…)***	Pré-remplissage aléatoire du **Sudoku** selon sa difficulté.
Bool	***isPlayableColumn(…)***	Vérifie si une action est jouable selon les cases de la colonne.
Bool	***isPlayableLine(…)***	Vérifie si une action est jouable selon les cases de la ligne.
Bool	***isPlayableSquare(…)***	Vérifie si une action est jouable selon les cases du carré.
Bool	***isPlayable(…)***	Vérifie si une action est jouable selon les 3 autres méthodes.
Void	***setCase(…)***	Place une valeur à la case (l,c).
Bool	***solveSudoku(…)***	Méthode phare sur la résolution du Sudoku. Par backtracking.

Void Sudoku::RandomFilling(int complexity) :

Cette méthode permet de pré-remplir la grille de Sudoku selon la difficulté (***complexity***) choisie. Pour cela, on utilise un switch. On randomise un nombre de cases aléatoires à pré-remplir dans un ***while*** jusqu'à l'obtention d'un nombre correct. Ce dernier dépend de la difficulté (1 -> + de 40 cases ; 2 -> 35 à 40 cases ; 3 -> 30 à 35 cases ; 4 -> 25 à 30 cases ; 5 -> 20 à 25 cases ; Autre/+ de 5 -> difficulté maximale : 20 cases max).

Ensuite, pour chaque case à remplir (selon ***nbToFill***), on choisit une case aléatoirement dans la grille et une valeur à donner (dépend de la taille ***size*** du **Sudoku**). On vérifie si c'est

jouable (cf. méthode ***isPlayable()***). Enfin, si c'est correct, on place cette valeur dans la case correspondante.

Bool Sudoku::isPlayable(int l, int c, int n) (et le package isPlayable(…) associé) :

Cette méthode permet de valider la jouabilité d'une action. Elle utilise 3 sous-méthodes vérifiant la validité de l'action selon la colonne (***isPlayableColumn(…)***), la ligne (***isPlayableLine(…)***) et le carré (***isPlayableSquare(…)***). Les deux premières sont extrêmement simples, on vérifie simplement selon les 8 autres cases (colonne ou ligne). Pour la dernière, on a besoin de représenter les carrés selon leur propre ligne et colonne. Par exemple : dans un Sudoku de taille 9x9, un carré sera sur positionné sur la ligne 0 à 2 des carrés, et de 0 à 2 pour les colonnes des carrés. On utilisera ensuite ces données pour calculer la position des cases (c,l) de ce même carré. (Voir code).

Bool Sudoku::solveSudoku(Sudoku& sudoku, int c, int l) :

Cette méthode permet de résoudre une grille de Sudoku par ***Backtracking***. Le principe est relativement simple : de manière récursive, on part du postulat qu'une action est valide (ici on affecte une valeur valide à une case). On continue ensuite de placer (aléatoirement) une valeur à la case suivante. Et on continue, case par case (ici dans le sens de lecture : de gauche à droite = ligne entière PUIS de haut en bas = ligne suivante). Si on arrive dans une situation où il faut remplir une case mais qu'aucune solution n'est possible, alors on abandonne cette affectation partielle et on retourne sur la précédente affectation et ses autres possibilités (***Backtracking*** = retour sur trace). Ainsi, on parcourt en profondeur un arbre de décision jusqu'à résolution du **Sudoku** (si elle existe). La description en détail du code est disponible en commentaire dans la méthode.

Main et fonctions de tests

Dans le main(), et pour chaque question, se trouve une fonction testX() où X correspond à la question associée.

Fonctions		
Type	**Nom**	**Explication**
Void	***Test1()***	Test de la création d'un **Sudoku** et de son affichage.
Void	***Test2()***	Test des méthodes de **Sudoku** qui valident une action.
Void	***Test3()***	Test de la création et pré-remplissage selon la difficulté.
Void	***Test4()***	Test sur la résolution d'un **Sudoku** (et selon sa difficulté).
Void	***Test5()***	Test avec des Sudokus de taille NxN.

Remarques diverses sur la réalisation du projet

- L'utilisation de fonctions tests au sein du main est très pratique pour réaliser des tests sur des objets. C'est une pratique que je connaissais mais que je n'utilisais que très peu -> A garder car cela permet de gagner beaucoup de temps sur la correction du non-fonctionnement/bug d'une partie du code en cours d'écriture.

- Il est évident que les temps de résolution sont très longs pour des grilles très grandes (j'ai noté : instantané pour du 9x9, 15 secondes pour du 16x16, bien plus pour du 25x25 et plus…).

- La méthode de résolution permet de résoudre et de vérifier en même temps la validité d'une grille (si la configuration initiale est résoluble ou non).

Pistes d'amélioration et suites possibles du projet

- Ajouter des vérifications diverses simples : taille positive du **Sudoku**…

- Une vérification supplémentaire : on ne vérifie pas si la taille du Sudoku est suffisante par rapport au nombre de cases à pré-remplir de manière aléatoire (même si on suppose qu'on ne travaille qu'avec des **Sudokus** de taille 9x9, 16x16x, 25x25, etc.).

- En plaçant un compteur dans la méthode de résolution par ***Backtracking***, on peut estimer cette avancée en l'affichant (en %) durant la résolution afin de suivre son évolution.

- Trouver un moyen de réduire le temps de résolution en utilisant des techniques de programmation, en utilisant des threads par exemple.

Conclusion / Expérience de programmation sur ce projet

Durant l'écriture de ce programme, j'ai renforcé mes connaissances sur la programmation en c++, notamment sur la gestion d'un tableau bidimensionnel avec des vectors. J'ai également revu la programmation par récursivité et sa capacité à résoudre un problème en calculant des solutions d'instances plus petites de ce même problème. Enfin, j'ai aussi pris plaisir à programmer ce sujet.