LIEUTAUD Emy

Résoudre un sudoku en Java

Un sudoku est un jeu en forme de grille, généralement une grille de taille 9x9 (avec des carrés 3x3). Le but est de remplir toute la grille avec des chiffres compris entre 1 et 9. Or pour insérer les chiffres dans la grille, il faut respecter quelques règles : un chiffre ne peut pas être contenu plus d'une fois dans une ligne, un carré ou une colonne. Autrement dit, chaque ligne, colonne et carré doit contenir tous les chiffres.

Objectifs :

- Écrire et implémenter en Java un algorithme permettant de résoudre tous les sudokus
- Optimiser l'algorithme au maximum afin d'avoir un temps de résolution minimal
- Améliorer la partie jeu

Amélioration de la partie jeu :

Dans la partie jeu, une grille est présentée à l'utilisateur. Il doit rentrer manuellement les chiffres afin de résoudre le sudoku. Pour cela, il est nécessaire de vérifier les saisies de l'utilisateur. En effet, une ligne et une colonne doit être comprise entre 0 et 8 tandis qu'une valeur est comprise entre 1 et 9. Ensuite, il faut vérifier que la valeur est correcte. Or cette valeur peut être correcte au début puis devenir fausse car au début de la résolution d'un sudoku, il y a peu de chiffre dans la grille donc il peut y avoir plusieurs possibilités pour une case. C'est pour cela que j'ai ajouté quelques lignes qui permettent à l'utilisateur de supprimer une valeur si celle-ci n'appartient pas à la grille de départ. Lorsque la grille sera remplie et correcte, le sudoku sera terminé !

Mise en œuvre d'un premier algorithme *solve()* :

J'ai tout d'abord réfléchi à plusieurs techniques permettant de résoudre un sudoku. Je me suis dit que la meilleure technique était d'utiliser les indices (qui représentent toutes les possibilités d'une case). J'ai donc créer un tableau d'objet. Ce tableau va me permettre de stocker tous les indices pour chaque case.
J'ai ensuite ajouté une méthode *convertObjectToList(Object obj)* qui va me permettre de convertir un objet en liste pour récupérer la liste d'indices pour une case donnée car on ne peut pas stocker des listes dans un tableau d'entiers.
Pour définir ces indices, j'ai écris la fonction *redefinirListeIndices(Object[][] maGrille)*. Elle va parcourir la grille et récupérer toutes les possibilités pour chaque case vide. Donc si nous avons une case vide dans la grille (égale à 0) et que la liste d'indices est de taille 1, cela veut dire que nous n'avons qu'une possibilité pour cette case et nous pouvons alors entrer cette valeur dans la grille. Cette méthode peut suffire pour les sudokus faciles. Je me suis ensuite aidée d'Internet afin de trouver les différentes techniques utilisées pour résoudre un sudoku.
Le *hidden single* est une méthode qui permet de trouver une valeur si deux colonnes ou deux lignes possèdent une valeur et qu'il n'y a qu'une possibilité de mettre cette valeur dans le carré car soit les deux autres places sur la ligne ou la colonne qui ne contient pas la valeur ne sont pas vides, ou alors parce qu'elles ne peuvent tout simplement pas avoir cette valeur.
Ensuite, si en parcourant une ligne ou une colonne on ne trouve pas la valeur en indice mais que la valeur se trouve seulement sur la case où l'on travaille et que la valeur est correcte, alors nous pouvons placer la valeur dans la grille. C'est ce que font les méthodes *verifierIndicesColonne(Object[][] maGrille, int i, int j, int valeur)* et *verifierIndicesLigne(Object[][] maGrille, int i, int j, int valeur)*.
La fonction *deductionIndices(Object[][] maGrille, int i, int j)* va chercher une ligne ou une colonne qui contient deux mêmes listes d'indices dans le même carré. Elle va ensuite regarder si sur cette ligne ou cette colonne (hors de son carré car la valeur est obligatoirement dans ce carré) si une case contient en indice cette valeur. Si aucune de ces cases vides ne peut contenir la valeur, alors nous pouvons entrer cette valeur dans le sudoku.

Ensuite, la méthode *testLigneEtColonne(Object[][] maGrille, int i)* va chercher une ligne ou colonne qui contient une valeur dans le même carré (c'est-à-dire deux case qui contiennent la valeur dans le même carré et sur la même ligne ou colonne). Nous allons ensuite regarder sur la ligne ou la colonne restante si la valeur peut se trouver que dans une seule case, sans regarder le carré trouvé précédemment car cette valeur y sera forcément. Si celle-ci ne se trouve qu'une seule fois, alors nous pouvons entrer cette valeur dans la grille à l'intérieur de la case trouvée.
Le *naked single* est une méthode qui va, au sein d'un carré, regarder si une valeur ne peut se trouver que dans une seule case.
Le *naked pair*, quant à lui, va parcourir une ligne ou une colonne. Si sur cette ligne ou colonne il trouve deux cases qui contiennent exactement les deux mêmes indices, alors il va pouvoir éliminer ces deux indices sur les autres cases de la ligne ou de la colonne. Si en éliminant des indices, il remarque qu'une case ne contient plus qu'un seul indice, alors il place cette valeur dans cette case afin de compléter la grille.
Enfin, le *naked triplet*, tout comme le *naked pair*, va chercher sur une ligne si une case contient 3 valeurs et que deux autres cases contiennent au moins 2 de ces valeur si elle possède 3 indices, ou une de ces valeur si elle possède 2 indices. Si deux cases sur 3 ont une liste d'indices de longueur 2, alors la 1ère case doit contenir 2 des 3 valeurs et la 2ème case doit contenir la 3ème valeur. Ces 3 valeurs forment alors un triplet et nous pouvons alors éliminer ces valeurs des listes d'indices des autres cases sur la même ligne ou colonne.

Cet algorithme est assez compliqué. De plus, il manque des méthodes telles que *swordfish*, *X-Wing* ou encore *XYZ-Wing*. Elles sont difficiles à comprendre et à implémenter. Il me faudrait donc encore plus de temps. Avec *solve()*, je ne peux donc pas résoudre tous les sudokus. De plus, nous remarquons que la grille de sudoku est parcourue un bon nombre de fois. Elle contient donc de nombreuses boucles *for* imbriquées ce qui n'est pas efficace. Le temps de calcul est donc mauvais.

Mise en œuvre d'un deuxième algorithme *solve2()* :

Il me fallait une méthode plus efficace, plus simple et qui puisse résoudre tous les sudokus. Je me suis donc une nouvelle fois documentée sur Internet et j'ai trouvé une autre méthode pour résoudre un sudoku, cette fois de manière récursive. Cette méthode va parcourir la grille. Dès qu'elle va trouver une case vide, elle va tester la case avec une valeur. Si cette valeur est correcte, alors elle va chercher une deuxième case vide et va refaire cette méthode. Si celle-ci retourne *true*, alors elle va continuer avec une autre case vide. Sinon, elle va remettre la case à zéro et refaire la fonction jusqu'à ce que la grille soit remplie.

Cet algorithme marche quelque soit la grille de départ. De plus, nous remarquons qu'il est beaucoup plus rapide que l'algorithme précédent et plus simple à implémenter et à comprendre.

Optimisation des méthodes :

Si pour la méthode *solve()* nous implémentons toutes les fonctions permettant de résoudre tous les sudokus et que nous optimisons toutes les fonctions utilisées, l'algorithme ne pourra pas être plus rapide que *solve2()*. En effet, *solve2()* est beaucoup plus simple et est beaucoup plus économe en terme de calcul. De plus, il n'appelle qu'une fonction : elle-même.