**ECRIRE EN LATEX**

**PAGE DE GARDE SUDOKU**Chef de projet et rédacteur de ce rapport : DIAZ Antoine  
Equipe : PERRET Isabelle, BOUDON Fanny  
Nom du projet : Algorithme de résolution de Sudoku

**Introduction**

Un Sudoku est un jeu composé d’une grille de 9x9 cases remplies de chiffres allant de 1 à 9. Chaque ligne, chaque colonne et chaque région 3x3 par lesquelles la grille est divisée ne doit comporter qu’une seule fois le même chiffre.

A la base le but était de laisser un utilisateur remplir une grille 9x9 de sudoku et de demander au programme de le résoudre.

Sachant désormais que nous sommes trois pour un projet qui ne demandait que deux personnes, nous avons étendu nos attentes.  
Le but actuel est de présenter un sudoku vide à l’utilisateur avec le choix de le remplir lui-même ou bien de laisser le programme en générer un avec solution. Ensuite, il a le choix entre essayer de le résoudre ou bien de laisser l’algorithme trouver la solution.  
En deuxième optique, nous chercherons à mettre plusieurs stratégies de résolution de sudoku afin de pouvoir comparer laquelle est la plus optimisée.

**Organisation du travail**

Nous avons choisi de faire le programme en java suite au cours reçu en L1 et L2 pour Fanny et moi-même, et après demande à Isabelle pour qui cela ne posait pas de problème.

Concernant la communication, nous avons mis en place un groupe WhatsApp pour les échanges faciles et les questions que nous avons.  
Nous avons ensuite mis en place un projet sur GitHub pour pouvoir nous partager nos codes et autres fichiers, comme le diagramme des taches mis sous forme de tableau à taches. [insert screen majé]

La division du travail s’est faite comme ceci :

FANNY ISABELLE ANTOINE

Le programme affiche une grille 9x9 vide avec deux choix : « Je crée mon sudoku » ou « Générer un sudoku aléatoire » :

* « Je crée mon sudoku » : L’utilisateur rempli les cases voulues, le programme l’arrête s’il met deux valeurs incompatibles (même ligne, même colonne, zone 3x3) en lui indiquant le problème.
* « Générer un sudoku » : Le programme génère un sudoku partiellement rempli et possédant une solution que l’utilisateur doit trouver (trouver les conditions de créations de sudoku)

L’utilisateur peut ensuite cliquer sur « Résoudre mon sudoku par algorithme » ou « J’essaye de résoudre le sudoku » :

* « Résoudre mon sudoku par algorithme » : Le programme affiche les nombres qu’il trouve au fur et à mesure. Il indique à la fin dans un compte rendu combien de temps chaque stratégie de résolution a mis pour trouver la solution, s’il y en a plusieurs (l’utilisateur n’a pas rempli assez de cases) ou s’il n’y en a pas du tout. L’utilisateur revient ensuite aux 2 choix initiaux
* « J’essaye de résoudre le sudoku » :
  + L’utilisateur essaye de résoudre le sudoku
  + Quand l’utilisateur veut repartir de zéro, il appuie sur un bouton « reset » qui remet le sudoku avec les chiffres initiaux
  + Quand l’utilisateur pense avoir finis, il clique sur le bouton « tester » et le programme test si sa solution est bonne (simplement en testant les lignes/colonnes/zones 3x3) :
    - Si sa solution est mauvaise il a le choix entre reset ses chiffres et repartir aux 2 choix initiaux
    - Si sa solution est bonne il repart aux 2 choix initiaux

Il doit y avoir dans tous les cas ci-dessus un bouton « retour au début » qui annule toute précédente action et fait repartir le programme à zéro ??

Chaque personne est totalement responsable de sa partie car nos codes ne sont pas encore liés. Nous préférons d’abord finir chaque partie de notre côté, ensuite les ajuster et finalement les optimiser ensemble.

Avec la roadmap de départ [insert roadmap] et la roadmap depuis mise à jour [insert roadmap majé].

**Antoine**

Pour trouver des idées d’algorithme j’ai cherché un peu sur internet et j’en ai trouvé vraiment beaucoup. Mais je me suis dit que pour le premier algorithme il fallait vraiment faire le moins optimisé possible pour voir la progression que l’on peut faire en termes de vitesse de résolution.

L’idée était donc d’écrire un algorithme de résolution qui teste chaque nombre l’un après l’autre dans chaque case l’une après l’autre jusqu’à ce que la grille soit valide.

Pour cela il fallait d’abord créer les méthodes évidentes mais essentiels à côté : le constructeur comprenant le sudoku, la méthode d’affichage du sudoku et une méthode test d’un sudoku rempli (ou partiellement rempli pour mon algorithme).

Cela a été la partie vraiment facile du premier algorithme de résolution.

Ensuite il a fallu s’attaquer au plus gros du problème : la méthode récursive de résolution. Pour chaque case de la grille 9x9 non vide, on teste chaque nombre de 1 à 9 pour voir s’il valide les conditions d’un sudoku (pas dans ligne, colonne et bloc 3x3), s’il valide les conditions on continue à la case suivante et ainsi de suite jusqu’à la fin de la grille. Si en revanche on ne trouve pas de nombre validant les conditions alors il faut revenir en arrière pour tester l’incrémentation de la case d’avant et ainsi de suite. Si finalement on revient au départ et qu’aucun nombre ne convient, alors il n’y a pas de solution.

Jusqu’ici rien de bien compliqué si ce n’est un peu de récursivité.

Maintenant il a fallu modifier le programme existant afin de correspondre aux attentes plus complexes de notre projet, ce qui m’a pris énormément de temps. A savoir :

* Montrer les nombres au fur et à mesure : pour cela, on affiche la grille à chaque itération de la première solution (aucun intérêt à le faire pour les autres solution)
* Faire en sorte que l’algorithme continue à chercher les autres solutions même après résolution du sudoku. Pour cela, l’algorithme affiche la solution et traite ensuite l’itération comme fausse pour pouvoir continuer à chercher des solutions jusqu’à avoir tester toutes les possibilités. L’algorithme est maintenant capable d’énoncer le nombre de solution et les solutions elles-mêmes si voulu.
* Donner le temps de résolution pour la première solution (je ne juge pas utile de donner les autres étant donné que cela serait faussé puisque la moitié du travail est déjà faite avec la première solution).

**Fanny**

**Isabelle**

**Conclusion**

Faut continuer en suivant la roadmap (majé ?)