Solveur et générateur de sudoku

MOUSSA MOHAMED Abdoulkader

Janvier 2023

1 Introduction

Ce rapport rentre dans le cadre de l'UE **Low-Level Programming** du semestre 1 qui a pour objectif de nous apporter les connaissances et compétences de base nécessaires pour programmer en langage C.

Nous avons, dans le cadre de cette UE, eté amené à programmer un solveur et un générateur du fameux jeu **sudoku**. Le solveur sudoku permet de résoudre des grilles de différentes tailles (4, 9, 16, 25, 36, 49, 64) en donnant une ou toutes les solutions possibles. Le générateur, quant à lui, permet de créer des grilles de sudoku de différentes tailles. Il peut également créer des grilles qui ont une unique solution.

Dans un premier temps, nous verrons les stratégies appliquées dans le solveur sudoku , ce qui lui permet d'avoir une meilleure performance. Dans un second temps, nous verrons comment notre générateur génère des grilles(unique ou pas) et cela en temps raisonnable.

2 Solveur sudoku

Dans cette partie, nous verrons les stratégies appliquées dans le solveur sudoku après le *homework-5* et qui lui permettent d'avoir une meilleure performance :

* **Réduction de la complexité des heuristique *cross-hatching* et *lone-number* :**

  Au *homework-5*, les heuristique *cross-hatching* et *lone-number* avaient une complexité de O(n^2). Or, diminuer cette complexité est largement faisable. Nous avons donc réduit cette complexité et elle est desormais en O(n).

* **Reduction de la complexité de la fonction *grid_ choice()* :**

  La fonction *grid_ choice()* retourne une couleur se trouvant dans le plus petit ensemble de couleur de toute la grille. La complexité temporelle de cette fonction a eté reduite passant de O(c * n^2) à O(n^2). (c variant de 1 à n).

* **Ajout d'une nouvelle hereustique *locked-candidate***

  Il existe plusieurs techniques avancées de sudoku(heuristique) qui peuvent être utilisés dans les grilles complexes pour éliminer des candidats et permettre ainsi une résolution plus rapide.

  Nous avons implémenté l'heuristique *locked-candidate* de type *pointing* .
  La régle est la suivante [1] : *Si un candidat se trouve au moins deux fois sur une même ligne ou colonne uniquement, et à l'intérieur d'un même bloc, alors il peut être éliminé sur toute cette ligne ou colonne à l'extérieur du bloc.*

  .

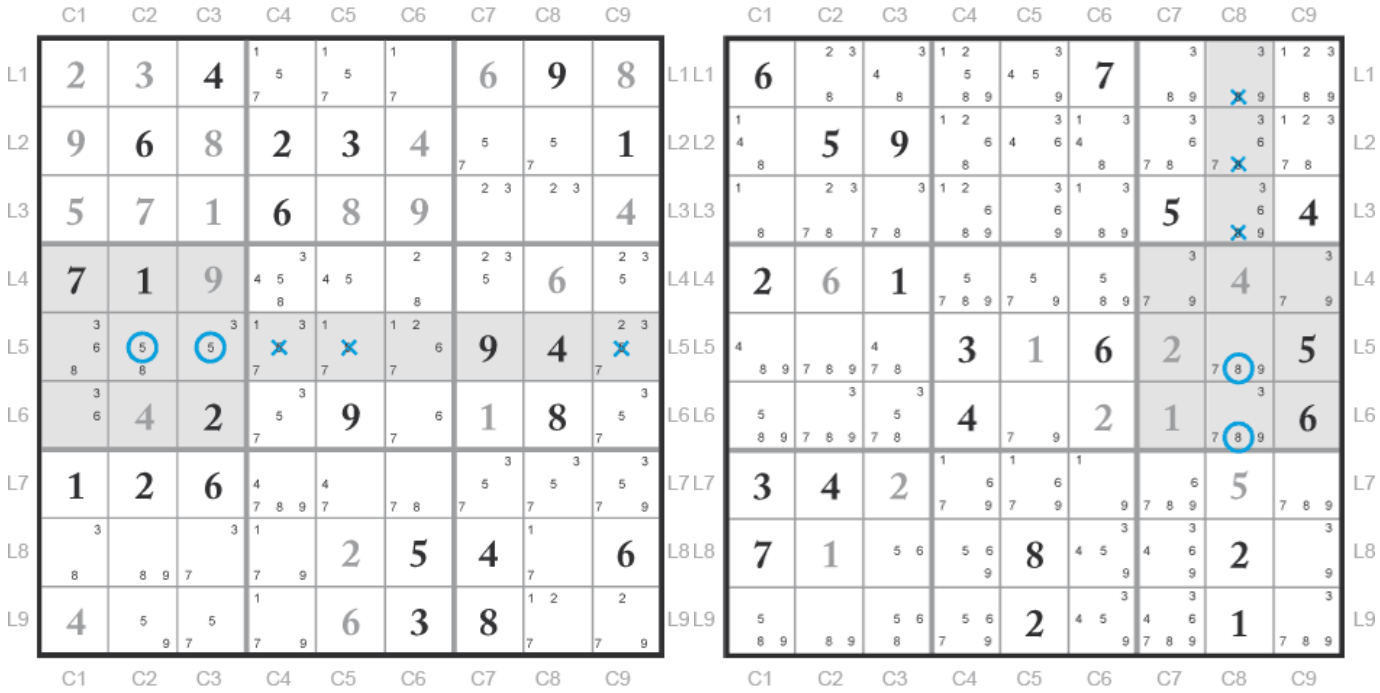

FIGURE 1 – *locked-candidate-pointing* sur une ligne et colonne[1].

Cette heuristique n'a pas beaucoup d'effet sur les grilles de niveau facile ou de petite taille. Mais a beaucoup d'effet sur les grilles de niveau difficile ou de grande taille.

Nous avons testé cela. Sans l'heuristique *locked-candidate*, notre solveur met plus de 5 minutes à résoudre la grille 64x64-11.sku de niveau 4. Et ne met que 4 secondes avec cette dernière.

∗ **Utilisés toutes les heuristiques en même temps peu être très couteux en temps :**

Nous avons maintenant implémenté plusieurs heuristiques. En les utilisant toutes en même temps, nous avons une complexité temporelle trés importante. Or, notre principal objectif est d'avoir une complexité temporelle la plus minimale possible. On doit donc se servir stratégiquement de nos heuristiques. La figure 2 ci-dessous illustre cela.

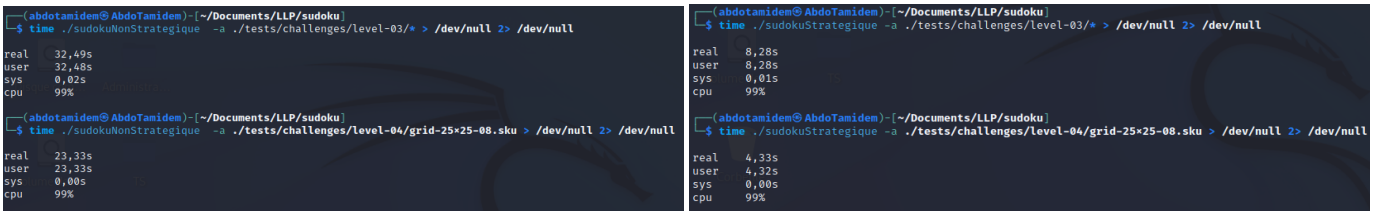

FIGURE 2 – heuristique utilisés non strategiquement vs strategiquement

Voici les choix qui ont eté fait pour se servir stratégiquement de nos heuristiques :

— **Ne pas utiliser l'heuristique *hidden-subset* :**
Nous avons constaté qu'avec ou sans l'heuristique **hidden-subset**, notre solveur sudoku prend approximativement le même temps pour résoudre une grille. Comme cela ne nous fait pas gagner du temps, nous avons donc decider de ne pas l'utiliser.

— **Appliquer l'heuristique *naked-subset* uniquement si la grille a changé après application des heuristique *cross-hatching* et *lone-number***

— **Moins utiliser l'heuristique *locked-candidate* :**
Nous avons remarqué que restreindre l'utilisation de l'heuristique *locked-candidate* fait gagner du temps à notre solveur. C'est-à-dire l'appliquer sur la grille uniquement si la grille n'a pas changé après application des autres heuristiques.

3 Générateur sudoku

3.1 Le solveur sudoku légèrement modifié

Pour pouvoir génèrer une grille, nous avons besoin d'un solveur sudoku qui :

— nous permet de savoir si une grille a **une** ou **plus d'une** solution.

— retourne une des solutions trouvées.

Or, pour une grille donnée, notre solveur sudoku :

— nous permet de savoir le nombre de solutions exactes.

— affiche **une** ou **toutes** les solutions.

Alors, pour pouvoir génèrer une grille en temps raisonnable, nous avons faits une légére amélioration sur une copie du solveur :

— Ajout d'une structure de type énumération **generator_t** qui contient comme valeur *mode_unique* et *mode_not_unique*. Ceci nous permet de savoir quel type de grille nous devons génèrer.

— Si la grille à génèrer est en *mode_not_unique*, c'est-à-dire qu'elle ne doit pas avoir obligatoirement une unique solution, alors dès qu'une solution est trouvée, notre solveur peut s'arrêter et retourner la solution trouvé.

— Si la grille à génèrer est en *mode_unique*, c'est-à-dire qu'elle doit avoir une unique solution, alors au bout de deux solutions trouver, notre solveur peut s'arrêter. Ceci car ce qui nous interesse vraiment c'est de savoir s'il ya **une** ou **plus d'une** solution.

— Le solveur ne doit absolument rien afficher..

— Ne pas appliquer l'heuristque *locked_candidates* sur la grille. Car cela ralenti énormement notre génèrateur.

3.2 Fonctionnement du génèrateur

3.2.1 Génèrateur en *mode_not_unique*

Pour génèrer une grille en *mode_not_unique*, notre générateur suit les étapes suivantes :

1. Créer une grille dont toutes les cases sont masquées(contiennent toutes les valeurs possibles). Puis on fixe une valeur dans une case aléatoirement. Ceci nous permet d'avoir une nouvelle grille à chaque fois.
2. On utilise notre nouveau solveur pour trouver une solution à cette grille. Notre nouveau solveur va s'arreter à la premiere solution trouvée .
3. On masque 40% des cases.

Temps d'execution en fonction de la taille de la grille

* Taille 4 : 0.00s
* Taille 9 : 0.01s
* Taille 16 : 0.01s
* Taille 25 : 0.03s
* Taille 36 : 0.18s
* Taille 49 : 0.34s
* Taille 64 : 1.10s

Analyse de la complexité temporelle

Soit n la taille de la grille à génèrer.

Etape 1 : O(n^2) car nous devons parcourir toutes les cases de la grilles pour pouvoir les masquer. Fixer une valeur à une case : O(1).

Etape 2 : Depend vraiment de n et du niveau de difficulté de la grille à résoudre.

Etape 3 : O(n).

3.2.2 Générateur en *mode_unique*

Pour générer une grille en *mode_unique*, notre générateur suit les étapes suivantes :

1. Créer une grille dont toutes les cases sont masquées(contiennent toutes les valeurs possibles). Puis on fixe une valeur dans une case aléatoirement. Ceci nous permet d'avoir une nouvelle grille à chaque fois.
2. On utilise notre nouveau solveur pour trouver une solution à cette grille. Notre nouveau solveur va s'arreter au bout de deux solutions trouvées.
3. On masque une case à la fois, aléatoirement.
4. A chaque fois qu'une case est masquée, nous lancons notre nouveau solveur sur la grille pour savoir combien de solutions nous avons.
5. Si la grille a une solution alors nous pouvons poursuivre le processus de l'étape 3. Sinon, nous remettons la valeur que nous avons retirée dans la grille.
6. Nous pouvons repeter le meme processus de l'etape 3 plusieurs fois jusqu'à masquer 40% des cases.

Les deux premières étapes sont identiques quelques soit la grille à génèrer.

Temps d'execution en fonction de la taille de la grille

* Taille 4 : 0.00s
* Taille 9 : 0.01s
* Taille 16 : 0.01s
* Taille 25 : 0.05s
* Taille 36 : 0.31s
* Taille 49 : 0.93s
* Taille 64 : (prend un peu de temps)

Analyse de la complexité

Soit n la taille de la grille à génèrer.

Etape 1 et 2 : Même complexité que les etapes 1 et 2 pour les grilles en mode_not_unique.

Etape 3 : O(1).

Etape 4 : Complexité du nouveau solveur sudoku

Etape 5 : O(1).

Etape 6 : O($\frac{n}{2}$).

4 Conclusion

Comme nous avions plusieurs heuristiques, il fallait faire des choix et les utiliser stratégiquement. Nous avons donc vu les differentes strategies appliquées dans notre solveur sudoku et qui lui permettent ainsi d'avoir une bonne performance.

Nous avons également vu comment notre programme arrivait à génèrer des grilles en temps raisonnable.

References

[1] "Régle locked candidates". (), [Online]. Available: https://sudoku.megastar.fr/blog/2018/05/31/reductions-locked-candidate/.