Rapport : Projet de Sudoku

Vieira Do Vale Thomas

November 23, 2023

Sommaire

1 Impélémentation de Gird Parser

Lors du deuxième TD, nous avions pour tâche d'implémenter un *grid_parser*. Le principe de se parser consiste à prendre une grille de sudoku stockée dans un fichier et à la stocker dans une structure de données appelée "grid". Dans un premier temps, j'ai mis en œuvre les fonctions utilitaires demandées, ce qui s'est avéré relativement simple. En revanche, l'implémentation du parser à poser des défis plus complexes.

La principale difficulté réside dans la gestion des nombreux cas différents qui peuvent se présenter. Il est impératif de transformer correctement la grille standard de base, mais aussi de gérer les grilles comportant des espaces superflus, des lignes vides, des commentaires, et même les cas où la grille fournie n'est pas valide. En effet, si le nombre de colonnes ou de lignes est incorrect, ou s'il y a un caractère incorrect, une alerte doit être générée.

Ma stratégie pour implémenter le parser a consisté à commencer par créer une fonction capable de parser correctement une grille normale. Ensuite, j'ai récupéré les différentes grilles que nous devions être en mesure de parser. J'ai soumis chaque grille à mon parser, puis j'ai corrigé les problèmes qui se présentaient. Cependant, cette approche a abouti à un code qui réussissait tous les tests, mais qui manquait de lisibilité, car il n'était pas structuré de manière claire.

Face à cette problématique, j'ai décidé de recommencer le travail à partir de zéro. Pour cela, j'ai pris l'initiative d'imprimer le code afin de l'étudier et de déterminer quelles parties pouvaient être regroupées de manière plus cohérente. J'ai également opéré des choix de conception différents. Dans ma première version, j'utilisais une première boucle pour récupérer la première ligne, puis je passais à une double boucle for pour gérer la taille des lignes et des colonnes. Dans ma nouvelle implémentation, j'ai opté pour une seule boucle while en fonction du caractère récupéré, me permettant ainsi d'utiliser des instructions switch case. Cette modification rend plus clair le traitement de chaque cas.

Un autre changement majeur visant à améliorer la lisibilité de mon code a été d'abandonner la gestion d'erreur à chaque cas. Auparavant, je libérais la mémoire, définissais une erreur à true, et fermait le fichier à chaque cas de sortie. Pour éliminer cette répétition de code, j'ai eu recours à des instructions "goto".

2 Implémentation de Lone Number

Dans cette section, nous aborderons l'implémentation de l'heuristique du "lone number". Le principe de cette heuristique est de déterminer si une couleur n'est présente que dans une seule cellule. Pour ce faire, j'ai commencé par mettre en place une implémentation naïve de cet algorithme. La méthode consiste en une boucle qui parcourt tous les indices. À l'intérieur de cette boucle, nous examinons l'ensemble de la sous-grille pour vérifier si l'indice apparaît plus d'une fois. Si c'est le cas, nous ne faisons rien ; sinon, nous parcourons à nouveau la sous-grille pour identifier l'unique cellule contenant l'indice, puis nous supprimons les autres couleurs présentes dans cette cellule. Cette approche a une complexité de $O(n^2)$, avec n représentant la taille de l'entrée.

Il m'est apparu évident qu'il existait des possibilités d'amélioration significative de cet algorithme. Ma première idée a été de stocker la première occurrence de chaque indice, afin d'éviter de refaire un deuxième parcours de la sous-grille. Cependant, cette amélioration n'a pas modifié la complexité. Il devenait donc impératif de trouver une solution bien plus efficace. Après réflexion, il est apparu possible de réaliser l'algorithme en $O(n)$, toujours avec n comme taille de l'entrée. Pour ce faire, il était nécessaire de repérer tous les "lone numbers" en un seul passage, puis de les supprimer dans un deuxième passage, sans que ces deux passages soient imbriqués.

L'idée pour repérer les "lone numbers" consiste à créer deux sous-ensembles : le premier, que nous appellerons judicieusement *first_appearance*, pour stocker les éléments lors de leur première apparition, et le deuxième, appelé *second_appearance*, pour stocker les éléments lors de leur deuxième apparition. Pendant notre premier passage, nous remplissons ces sous-ensembles, mais cette fois-ci en examinant les cellules plutôt que les indices.

Pour remplir *first_appearance*, nous effectuons l'union de *first_appearance* avec la sous-grille examinée. En procédant ainsi, nous nous assurons de rassembler tous les éléments déjà vus une fois. Pour *second_appearance*, nous devons identifier les éléments présents à la fois dans *first_appearance* et dans la sous-grille examinée, ce qui nécessite une intersection entre les deux ensembles. Il est crucial de réaliser cette opération avant l'union, sinon nous considérerions que tous les éléments de la sous-grille ont déjà été vus deux fois, ce qui peut ne pas être le cas.

Une fois les deux sous-ensembles remplis, les "lone numbers" sont les éléments présents dans *first_appearance* mais absents de *second_appearance*. Nous les identifions donc en effectuant l'intersection entre les deux ensembles. La dernière étape consiste à trouver les cellules contenant ces "lone numbers". Nous parcourons la grille en faisant l'intersection entre la cellule examinée et l'ensemble des "lone numbers". Si cette intersection n'est pas triviale, nous remplaçons la cellule par cette intersection, qui représente notre "lone number". Il est essentiel de ne pas effectuer ce remplacement si l'intersection est vide ou si la cellule

examinée est un singleton, car cela évite de créer des changements en boucle, et donc que le programme ne se termine jamais.

3 Implémentation de Hidden Subset

Dans cette section, nous explorerons l'implémentation de l'heuristique du "hidden-subset". Le principe fondamental de cette heuristique consiste à rechercher un sous-ensemble de taille n présent dans n cellules. Il est important de noter que ce sous-ensemble n'est pas nécessairement complet dans toutes les cellules. Lorsqu'un tel sous-ensemble est identifié, il devient possible de supprimer toutes les autres couleurs présentes dans les cellules où l'intersection avec l'ensemble n'est pas vide.

L'implémentation de cette heuristique s'est avérée être la plus complexe parmi toutes. J'ai fait une première version qui respectait strictement l'heuristique. Mais le code de cette heuristique n'était absolument pas performant. Lors de différents tests, je me suis rendu compte que si je restreignais les sous-ensembles cachés à partir d'une cellule, je pouvais améliorer grandement l'efficacité. Le sous-ensemble recherché est donc contenu obligatoirement dans une cellule, mais n'est pas obligé d'être la cellule entière.

Mon nouvel algorithme fonctionne de la manière suivante : pour chaque cellule, deux opérations distinctes sont effectuées. Premièrement, nous comptons le nombre de cellules ayant une intersection non vide avec la cellule en cours d'examen, en stockant dans une liste les cellules qui satisfont cette condition.

La deuxième opération consiste à vérifier si le nombre de cellules trouvées est équivalent au nombre de couleurs de la cellule en cours d'examen. Dans ce cas, nous sommes autorisés à supprimer les éléments qui ne font pas partie de l'intersection de ces cellules avec la cellule en cours. La conservation des cellules dans ce scénario particulier permet d'éviter de parcourir l'ensemble complet de la sous-grille. Cette approche optimise le processus en se concentrant uniquement sur les cellules pertinentes, améliorant ainsi l'efficacité globale de l'algorithme.

4 Algorithme de génération de sudoku

Nous allons maintenant explorer l'implémentation du générateur de sudoku. Ma stratégie d'implémentation a consisté d'abord à créer un générateur fonctionne sans le mode unique, puis à ajouter le mode unique par la suite. Il existe plusieurs approches pour générer une grille. On peut partir d'une grille vide et faire des choix aléatoires pour la remplir en s'arrêtant à notre ratio de remplissage, dans notre cas, il est demandé de remplir à hauteur de 75%. Une autre

possibilité est de résoudre une grille vide, puis de retirer des cases. J'ai préféré opter pour la deuxième option, car il m'a semblé plus simple d'implémenter l'unicité dans ce cas.

Maintenant que j'ai défini ma conception, il fallait trouver comment générer une grille remplie de manière aléatoire. Le moyen le plus simple était de réutiliser la fonction de backtrack que nous avons codée pour le solveur. Pour ma première version, j'ai envisagé de partir d'une grille vide et d'utiliser le backtrack pour la résoudre. Cependant, le problème était que cette fonction faisait des choix qui n'étaient pas aléatoires. Il fallait donc trouver un moyen d'ajouter cette composante sans altérer le fonctionnement du backtrack. J'ai donc créé une fonction *choice_random* qui choisit au hasard une case qui n'est pas un singleton, puis fait un choix aléatoire pour la valeur de la case. Cette implémentation pose des problèmes de complexité. Il prend beaucoup trop de temps pour générer des grilles de taille 64, donc il est nécessaire d'améliorer la vitesse du backtrack et de lui fournir une grille plus facile à résoudre.

Pour améliorer le backtrack, la meilleure approche consiste à faire comme dans *grid_choice*, mais en effectuant un choix aléatoire. Le moyen le plus simple est de simplement modifier la fonction *grid_choice* en remplaçant *colors_rightmost* par *colors_random* au moment du choix. Il peut sembler étonnant que cela ne modifie pas les performances du solveur. En réalité, il est possible que cela améliore ou ralentisse le *mode_first*, mais nous ne pouvions pas garantir que le choix le plus à droite était optimal. Ainsi, prendre une option au hasard ne nous fait rien perdre. Pour le *mode_all*, on explore toutes les possibilités de toute façon, donc prendre le premier choix de manière aléatoire ne change rien.

Pour donner à la grille plus de contraintes, il est nécessaire de faire des choix avant de le soumettre au backtrack. La méthode la plus simple est de résoudre un bloc, une ligne et une colonne. Ainsi, on est sûr d'avoir toujours une grille cohérente, mais on réduit considérablement les possibilités du backtrack. Pour effectuer ces choix, j'ai décidé de commencer par remplir le premier bloc (celui en $(0, 0)$), puis de remplir les lignes et blocs en appliquant les heuristiques entre chaque choix. Les choix sont évidemment faits de manière aléatoire.

Malgré tout cela, il arrive des moments où le backtrack ne trouve tout simplement pas de solution en un temps raisonnable. La solution trouvée est de lancer le backtrack, mais s'il dépasse un certain temps, alors on abandonne et on relance. Pour implémenter cette méthode, on utilise le temps, stocké lors du lancement de la fonction. À chaque récursion du backtrack, on vérifie si on a dépassé le temps défini. Cependant, cela ne doit se produire que lorsqu'on le lance depuis le générateur. Pour cela, j'ai ajouté un mode appelé *mode_generate*. Pour déterminer le temps optimal, des tests ont été réalisés avec différents intervalles. Après mes essais, j'ai trouvé que trente secondes étaient le temps idéal.

Maintenant, avec une grille résolue en main, il faut supprimer un certain nombre

de cases. La première version utilisait simplement une boucle *for* qui retirait un élément calculé aléatoirement. Pour déterminer la case à retirer, on imagine la grille comme une seule grande ligne. On choisit donc un chiffre entre 0 et $size * size$, puis on calcule à quelle case cela correspond dans la grille. Une fois cela fait, nous avons le générateur fonctionnant pour le mode unique. Pour passer en mode unique, il suffit de modifier la deuxième partie. La solution que j'ai choisie pour faire fonctionner le mode unique est de vérifier à chaque fois que je retire un élément que la solution est unique. Si ce n'est pas le cas, je rétablis le choix. Je pourrais utiliser le *mode_all*, mais cela prendrait du temps pour rien parfois. Je veux m'arrêter dès que j'ai trouvé deux solutions. J'ai donc ajouté un nouveau mode, le *mode_unique*. Avec ce nouveau mode, il suffit de lancer le backtrack avec cette option, puis s'il trouve deux solutions, alors je rétablis le choix.

Cet algorithme a des performances acceptables, bien que variables. Il se peut qu'il trouve une solution dès la première tentative de génération, mais il se peut aussi qu'il ne trouve jamais de solution, bien que cela soit peu probable. Malgré cela, le temps moyen d'exécution du programme reste acceptable.

5 Amélioration du code

Le code pouvait être amélioré à plusieurs niveaux, tant en termes de qualité que de performance.

En ce qui concerne la qualité, j'ai utilisé l'outil Valgrind pour vérifier l'absence de fuites de mémoire dans divers scénarios. Lorsque des fuites de mémoire étaient détectées, j'analysais l'origine de l'erreur et tentais de comprendre le moment opportun pour libérer la mémoire. Malheureusement, cette tâche nécessitait souvent plusieurs tentatives, en particulier dans la phase de backtrack. En effet, lors du backtrack, une copie de la grille était transmise, et le backtrack renvoyait également une grille, qui pouvait être identique à celle fournie ou créée par le backtrack. Il était donc essentiel de faire preuve d'une grande prudence à ce niveau.

Pour optimiser les performances, j'ai consacré du temps à améliorer
l'algorithme des différentes heuristiques et fonctions. Deux grandes parties ont été examinées : les heuristiques et la génération. Pour les heuristiques, j'ai lancé le solveur sur toutes les grilles disponibles dans le dossier *grid_solver*. Pour pouvoir vérifier qu'une amélioration était fonctionnelle, il fallait que je vérifie deux choses, que je trouve toujours le bon nombre de solutions, et que j'étais plus rapide qu'avant. Pour me simplifier la tâche, j'ai écrit deux scripts bash, et vérifiant que je trouve les mêmes nombres de solutions que votre programme, et l'autre qui lance les tests sur toutes les grilles. Si je trouvais le bon nombre de solutions, et que le temps était plus rapide sur toutes grilles, alors les améliorations était validée. Mais si elle n'améliorait pas le temps sur toutes les

grilles et même ralentissait sur certains, il fallait comparer le temps gagner et perdu. Un cas particulier était celui des heuristiques "hidden subset". Il était possible de définir une condition sur la taille des ensembles à examiner, mais il était crucial de déterminer le moment optimal pour effectuer cette vérification. Pour ce faire, j'ai réalisé des tests empiriques en suivant la même méthodologie que précédemment. Les résultats ont montré qu'aucune amélioration significative n'était obtenue en ne considérant pas toutes les possibilités.

Pour augmenter la vitesse du générateur, le paramètre principal était le temps accordé au backtrack pour générer une grille. Étant donné que le temps de génération d'une grille était aléatoire, la comparaison entre deux exécutions n'était pas suffisante, nécessitant une moyenne sur dix exécutions pour évaluer les différentes options. J'ai constaté que la meilleure durée pour mon programme était de vingt secondes.