Projet de Programmation C

BESLAY Cyril

13 Novembre 20009

Règles du Sudoku et Ensembles préemptifs

Une grille de Sudoku est une grille de taille $n^2 \times n^2$. Cette grille est séparée en n^2 blocs de taille $n \times n$ (voir Figure 1).

Une solution, pour être valide, doit vérifier cette définition : "Chaque colonne, chaque ligne et chaque bloc doit contenir toutes les couleurs de l'ensemble." Un ensemble est constitué de couleurs disponibles pour remplir la grille. Par exemple, l'ensemble d'une grille 9×9 est l'ensemble des chiffres $\big[1, 2, 3, 4, 5, 6, 7, 8, 9\big]$.

Dans le cas de grilles de taille supérieure à 9×9, on introduit des lettres.

Pour résoudres les grilles, on utilise les ensembles préemptifs. Un ensemble préemptif est un ensemble composé d'éléments de l'ensemble de la grille. Seuls les éléments étant des solutions possibles sont dans cet ensemble. Chaque cellule de notre sudoku contient donc un ensemble préemptif.

Lorsque l'on applique des algorithmes qui suppriment les choix impossibles dans les ensembles préemptifs. Le sudoku est résolu quand tous les ensembles ne contiennent plus qu'une seule solution.

Pour notre projet, nous avons utilisé des ensembles préemptifs sous la forme d'entiers de 32 bits. Les 24 premiers bits correspondent aux couleurs, 1 pour possible, 0 pour impossible. Le bit 25 sert à définir si la couleur était définie dans le Sudoku de départ.

De cette manière, les calculs sur les ensembles préemptifs sont simplifiés et demandent peu d'opérations (calculs en binaire).

9	4	3	6	5	7	8	2	1
7	8	1	2	3	9	5	4	6
6	2	5	1	4	8	7	9	3
4	5	8	3	1	6	2	7	9
3	7	2	5	9	4	1	6	8
1	9	6	7	8	2	4	3	5
5	6	7	9	2	1	3	8	4
2	1	4	8	6	3	9	5	7
8	3	9	4	7	5	6	1	2

Figure 1

Heuristiques

Une heuristique est une méthode qui donne une solution réalisable mais pas optimale, contrairement aux algorithmes. Les heuristiques sont utilisées pour des problèmes NP-complets.

Nous divisons notre grille de Sudoku en sous-grilles (par ligne, colonne et bloc) pour éxecuter les heuristiques sur toutes les sous-grilles.

Ici, le rôle des heuristiques et de supprimer de chaque sous-grille les solutions impossibles en appliquant des méthodes différentes que j'explique dans la suite.

Cross-hatching

L'heuristique du cross-hatching consiste juste à identifier toutes les cellules qui sont résolues (une seule couleur) et enlève la couleur de toutes les autres cellules de la sous-grille. Par exemple, si la sous-grille est $\{1234, 2, 123, 14\}$, alors cette heuristique retire tous les 2 de la sous-grille et donne $\{134, 2, 13, 14\}$.

J'ai codé cette heuristique en utilisant deux boucles *for*. La première parcoure la sous-grille à la recherche de singletons (une seule couleur dans le pset). Lorsqu'elle en trouve un, la deuxième boucle supprime cette couleur de tous les autres psets grâce à la fonction

Cette heuristique est en O(n^2).

Lone-number

L'heuristique du lone-number veut que si une couleur est représentée une unique fois, alors c'est la bonne couleur. Par exemple, si la sous-grille est $\{123, 2, 123, 124\}$, alors cette heuristique valide l'ensemble qui contient la seule occurence de 4 et donne la sous-grille $\{123, 2, 123, 4\}$.

J'ai codé cette heuristique en utilisant la fonction *exclusive_union*. Dès qu'on recontre un pset, on le compare aux autres en enlevant à chauqe fois les couleurs qui apparaissent. Si une couleur n'était présente que dans le pset initial, elle se retrouve comme singleton à la fin et on peut donc la valider.

Cette heuristique est en O(n^2).

N-possible ou Naked subset

L'heuristique du N-possible implique que si un nombre N de couleurs est représenté N fois dans une sous-grille, alors ce sont les seuls endroits où ces couleurs sont possibles, et on peut donc les supprimer des autres ensembles. Par exemple la sous-grille $\{156, 56, 2, 56\}$ devient $\{1, 56, 2, 56\}$ car la paire 56 apparaît deux fois.

Pour implémenter cette heuristique, j'ai ajouté une nouvelle fonction aux psets, *pset_count*. Cette fonction compte le nombre de couleurs présentes dans un pset. Cela évite la répétition de codes, puisque cette fonction est souvent utilisée plus tard.
Naked subset compte le nombre N de couleurs dans un pset, et si ce pset apparaît N fois dans la sous-grille, on enlève ces couleurs des autres pset.

Cette heuristique est en O(n^3).

Solveur

Cette partie explique le raisonnement utilisé pour résoudre les grilles.

Structure du solveur

La première partie de la fonction *grid solver* applique les heuristiques à toutes les sous-grilles. Les fonctions *enumerate...* servent à partager la grille en sous-parties. Une boucle *while* permet de résoudre la grille au maximum. Tant que des changements sont effectués sur la grille, on continue à appliquer les heuristiques.
La deuxième partie est le back-tracking expliqué ci-dessous. La combinaison des ces deux méthodes permet de résoudre toutes les grilles consistantes.

Back-tracking

Lorsque les heuristiques n'effectuent plus de changements, deux cas se présentent. Soit la grille est résolue et on peut sortir de la fonction. Soit il reste des cellules non résolues. Dans ce dernier cas, nous utilisons la technique de *back-tracking*. Cette méthode permet de finir de résoudre la grille en faisant des choix aléatoires.
Voici l'algorithme utilisé :

- sauvegarde de la grille
- choix du pset le plus petit non singleton
- choix d'une couleur
- trois cas se présentent alors :
  1. grille résolue $\Rightarrow$ retourne *true* et la grille résolue
  2. grille non résolue $\Rightarrow$ on refait un choix
  3. grille inconsistante $\Rightarrow$ on remonte et on fait un autre choix

Le problème est que si toutes les couleurs ont été utilisées sans succès dans un pset, plus aucune solution n'est possible. On arrête donc la recherche et on retourne *false*.

Pour le choix du pset dans le backtracking, j'ai choisi de prendre le dernier plus petit. En effet, ce parcours demande un peu de temps, mais raccourci au final le temps global de résolution. En prenant le plus petit pset, on limite le nombre de tests pour chaque niveau de back-tracking. En prenant le dernier, on accelère la résolution, puisque si on tombe sur un mauvais choix, le solveur s'en rendra compte plus vite.

Lors du choix de la couleur dans le pset, deux choix se présentaient, soit choisir au hasard, soit choisir le premier. Pour avoir des résultats plus constants, j'ai choisi la deuxième option, les deux étant équivalentes en terme d'efficacité.

J'ai aussi choisi de m'arrêter la résolution à la première solution trouvée (sauf pour la génération stricte, que j'expliquerais plus tard).

Génération de grilles

Nous voici donc au projet proprement dit : la génération de grille. Le but est de générer des grilles de Sudoku avec un taux de cases remplies entre 25% et 50%.

Principe général

La fonction *generate_grid* commence donc par initialiser une grille vide (avec tous les psets pleins). On choisit alors une couleur au hasard pour la dernière cellule et on résoud la grille. Grâce au backtracking, une grille aléatoire de taille voulue est créée. On supprime ensuite un nombre de cellules suffisant pour que la grille soit vraisemblable. De même, le choix des cellules à supprimer doit être aléatoire.

Option -s (unique solution)

L'option -s doit permettre de générer des grilles de la même façon que précedemment mais avec une contrainte supplémentaire : la grille ne doit comporter qu'une seule solution.

Pour cela, je stocke toutes les cases de la grille dans un tableau. Puis je choisis au hasard une case dans le tableau. Je supprime cette case dans la grille (après l'avoir sauvegardée). Je lance la fonction de résolution en qui retourne *false* s'il existe plusieurs solutions. Si cette grille n'admet qu'une seule solution, je continue, sinon je remets la cellule supprimée précedemment.

Cet algorithme s'arrête quand plus de 50% des cases ont été supprimées ou quand toutes les cellules ont été testées de cette manière (le tableau est vide).

Voici l'algorithme général de génération résumé :

1. on génère une grille aléatoirement
2. on initialise une liste de cases à effacer avec toutes les cases du Sudoku
3. on choisit au hasard une case dans cette liste (et on l'enlève de la liste)
4. on efface, dans la grille, la valeur dans cette case
5. si le Sudoku n'a pas une solution unique, alors on remet la valeur effacée
6. si la liste est vide ou si assez de cellules sont vides, c'est fini, sinon on va en 3

Implémentation

J'avais choisi au début une liste chaînée pour stocker les cellules à choisir au hasard. Mais le parcours de la liste prenait du temps. La gestion grâce à un tableau s'est avérée plus rapide et moins coûteuse en mémoire. Par contre, deux sauvegardes de la grille sont nécessaires et cela prend beaucoup de place.

Tests

J'ai effectué de nombreux tests notamment avec *time* et *gprof* pour l'efficacité du programme et *valgrind* pour la gestion de la mémoire.

J'ai analysé avec *gprof* les fonctions qui prenaient le plus de temps à l'éxécution. J'ai donc séparé mes heuristiques dans des fonctions pour pouvoir analyser leur efficacité.

C'est logiquement Cross-hatching qui prend le plus de temps. En effet, c'est l'heuristique qui agit le plus souvent. On rentre donc dans les boucles incluses plus souvent que pour les autres heuristiques. C'est ensuite Lone-number puis Naked-subset dans l'ordre de temps d'éxécution.

Bugs

L'éxécution avec l'outil *valgrind* m'a permis de chasser les fuites mémoires. Elle se situaient principalement au niveau des grilles temporaires. Ces grilles étaient allouées en mémoire, mais je ne libérais pas l'espace attribué. Une fois ce problème résolu, aucun bloc mémoire n'était perdu.

Certaines grilles (notamment des grilles de taille 25) sont longues à résoudre. Il suffit qu'un choix de backtracking soit mauvais au début et le nombre de tests à faire explose.

Suivant le choix de la couleur dans le back-tracking, le temps de résolution change pour certaines grilles. Par exemple pour la grille 25x25-02 fournie avec les tests, le fait de prendre la dernière couleur du pset au lieu de la première fait passer le temps de résolution de quelques minutes à 20 sec. Mais d'autres grilles sont du coup plus longues à résoudre. Cela dépend de la structure de la grille initiale.

Optimisation

J'ai rajouté un test d'inconsistance à chaque tour d'heuristiques, ce qui évite de faire tourner le programme pour rien et fait gagner du temps.

J'ai aussi essayé d'améliorer les heuristiques en rajoutant quelques tests qui évitent des boucles pour rien.

Lors de la génération stricte, la fonction de résolution s'arrête dès qu'elle trouve une deuxième solution, ce qui raccourcit le temps de génération.

Conclusion

Ce projet nous a permis d'apprendre le developpement de A à z d'un programme. Nous avons construit le solveur en implémentant les parties au fur et à mesure, en pouvant améliorer chaque partie en fonction des demandes.

Le projet en lui-même nous a forcé à chercher la méthode à utiliser, les algorithmes et l'implémentation les plus efficaces. Chacun a donc construit un générateur différent, alors que l'ensemble du programme reste semblable pour tout le monde.