Création d'un solveur et d'un générateur de sudoku

Marion CANDAU

13 novembre 2009

Table des matières

1 Introduction **3**
1.1 Règles du Sudoku 3
1.2 Ensembles préemptifs 3

2 Heuristiques **4**
2.1 Heuristique du "cross-hatching" 4
2.1.1 Exemple 4
2.1.2 Algorithme du programme 5
2.2 Heuristique du "lone-number" 5
2.2.1 Exemple 5
2.2.2 Algorithme du programme 6
2.3 Heuristique du "naked subset" 6
2.3.1 Exemple 6
2.3.2 Algorithme du programme 7
2.4 Heuristique de la "disjoint chain" 7
2.4.1 Exemple 7
2.4.2 Algorithme du programme 8
2.5 Heuristique du "hidden subset" 8
2.5.1 Exemple 8
2.5.2 Algorithme du programme 8

3 Fonctionnement du solveur **9**
3.1 Heuristiques 9
3.2 Back-tracking 10

4 Génération des grilles **11**

5 Implémentation **12**
5.1 Structures de données 12
5.2 Découpage du code 12
5.3 Optimisation 12

6 Tests **13**
6.1 Utilisation de time 13
6.2 Utilisation de gprof 13
6.2.1 Résolution des grilles 13
6.2.2 Génération des grilles 14

1 Introduction

1.1 Règles du Sudoku

Une grille de sudoku standard est composée de 9 régions, 9 colonnes et 9 lignes. Le but du jeu est de placer la totalité des chiffres de 1 à 9 sans exception dans chaque ligne, dans chaque colonne et dans chaque région. Pour les grilles de tailles plus petites ou plus grandes, le but du jeu est le même, seulement la taille de la grille est différente et donc on place les chiffres de 1 à 4 dans le cas d'une grille de taille 4, les chiffres et les lettres de 1 à 9 et de A à G, pour une grille de taille 16, et les chiffres et les lettres de 1 à 9 et de A à P, pour un sudoku de taille 25. On a seulement le chiffre 1 pour une grille de taille 1. L'image 1.1.1 donne un exemple pour chaque taille de grille.

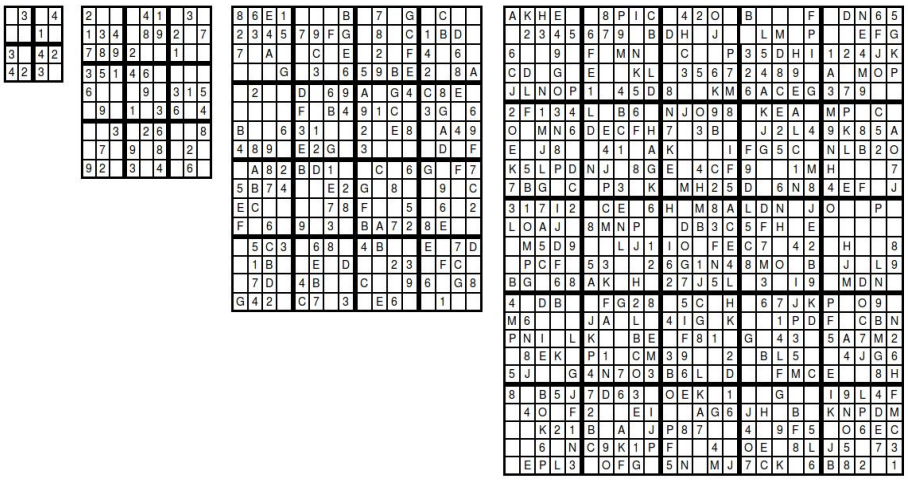

Fig. 1.1.1 – Exemple de sudokus de tailles 4,9,16 et 25

1.2 Ensembles préemptifs

Un ensemble préemptif est composé des éléments de l'ensemble $\{1, \cdots, 9\}$ (pour des sudokus de taille 9). Ces éléments représentent les couleurs qui sont encore possibles dans une cellule et seulement ceux-là, les éléments de l'ensemble $\{1, \cdots, 9\}$ qui sont pas dans l'ensemble préemptif sont exclus de la cellule. "2457" est un exemple d'ensemble préemptif, les couleurs '2', '4', '5' et '7' sont encore possibles dans la cellule, et les autres sont impossibles. Ces ensembles sont codés sur 32 bits de la manière suivante :

- les bits 0 à 24 servent à coder la présence de la couleur dans l'ensemble (ils sont mis à '1' dans ce cas) ou son absence (bit à '0').
- le bit 25 sert à coder le fait qu'il s'agit d'une cellule déjà remplie dans la grille initiale donnée par l'utilisateur.
- les bits 26 à 31 ne sont pas utilisés.

L'utilité d'utiliser des entiers codés sur 32 bits plutot que des chaines de caractères est le gain de mémoire et la facilité grâce à des fonctions définies dans le fichier preemptive_set.c d'utiliser ces entiers, notamment pour obtenir des intersections, des unions ou savoir si un ensemble préemptif est inclus dans un autre.

2 Heuristiques

Pour résoudre un sudoku, il existe différentes techniques. Ces techniques ne nous assurent pas de trouver la solution à coup sûr, mais elles résolvent certains sudokus dits "faciles". Pour les autres sudokus, elles permettent de diminuer les ensembles préemptifs et donc de s'approcher de la solution qui sera ensuite trouvée en utilisant le back-tracking. Dans ce solveur de sudoku, 5 heuristiques sont implantées, elles sont présentées dans les parties suivantes.

2.1 Heuristique du "cross-hatching"

L'heuristique du cross-hatching consiste à éliminer dans les ensembles préemptifs la couleur des cellules déjà résolues dans la sous-grille (colonne, ligne ou région).

2.1.1 Exemple

L'image 2.1.1 nous montre un exemple avec une ligne d'un sudoku 4×4 :

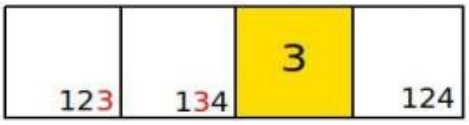

FIG. 2.1.1 – Exemple de cross-hatching

On peut alors éliminer des ensembles préemptifs les chiffres colorés et on obtient ainsi dans l'image 2.1.2 :

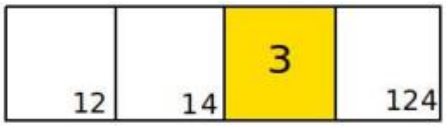

FIG. 2.1.2 – Exemple de cross-hatching

2.1.2 Algorithme du programme

L'algorithme d'une itération est le suivant : on teste si la couleur d'une case déjà résolue (c'est-à-dire avec seulement un candidat possible) de la sous-grille courante se trouve dans une autre case. Si oui, on enlève la couleur de cette autre case.

2.2 Heuristique du "lone-number"

L'heuristique du lone-number sert à faire appliquer la règle suivante : si une couleur est représenté une unique fois au sein d'une même colonne, ligne ou région alors cette couleur est la bonne.

2.2.1 Exemple

L'exemple de l'image 2.2.1 montre le principe de cette heuristique :

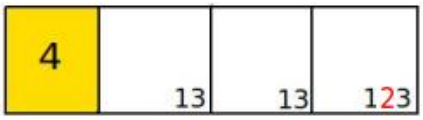

FIG. 2.2.1 – Exemple du lone-number

Le '2' est seulement dans la dernière case donc il ne peut être que là et donc à la fin de cette heuristique on obtient dans l'image 2.2.2 :

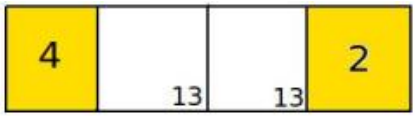

FIG. 2.2.2 – Exemple du lone-number

2.2.2 Algorithme du programme

On compte le nombre de fois qu'une couleur se trouve dans chaque case de la sous grille courante. Si cette couleur n'apparait qu'une fois, c'est un "lone-number", et donc la case dans laquelle cette couleur se trouvait, devient une case résolue avec pour couleur, la couleur du "lone-number".

2.3 Heuristique du "naked subset"

Le principe de cette technique est que si N cases ne contiennent que N fois les mêmes candidats il est certain que les N candidats se trouveront dans l'une de ces N cases. On peut donc supprimer ces candidats des autres cases. L'image 2.3.1 donne un petit exemple avec N=3.

2.3.1 Exemple

On remarque que les candidats '5','6','8'(en rouge) sont présents tous seuls dans 3 cases.

FIG. 2.3.1 – Exemple de naked-subset

On peut donc enlever ces candidats des autres cases. Ce qui nous donne après élimination des candidats dans les autres cases, la ligne 2.3.2 :

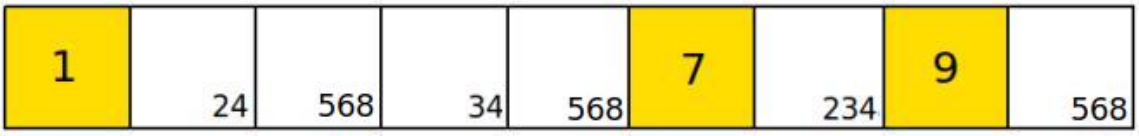

FIG. 2.3.2 – Exemple de naked-subset

2.3.2 Algorithme du programme

Pour chaque case qui n'est pas déjà résolue, on compte le nombre de cases de la sous grille courante qui ont exactement les mêmes couleurs possibles que cette case. Si ce nombre est égal au nombre de couleurs possibles de la case testée, alors nous sommes en présence d'un sous-ensemble nu. Et donc on enlève les couleurs trouvées, dans les cases où il n'y a pas le sous-ensemble nu.

2.4 Heuristique de la "disjoint chain"

Cette heuristique appelée également groupes isolés, ressemble beaucoup à la méthode du "naked subset", cette dernière étant un cas particulier des groupes isolés. La différence est que les N candidats ne sont pas tous présents forcément dans les N cases.

2.4.1 Exemple

Voyons un petit exemple à l'image 2.4.1 pour bien comprendre la différence. Le groupe isolé est identifié en rouge et on voit bien que tous les chiffres ne sont pas présents dans toutes les cases.

FIG. 2.4.1 – Exemple de disjoint-chain

On élimine ces candidats des autres cases ce qui nous donne à l'image 2.4.2 :

FIG. 2.4.2 – Exemple de disjoint-chain

2.4.2 Algorithme du programme

Pour chaque case qui n'est pas déjà résolue (notons la A), on compte le nombre de cases de la sous grille courante (notons les B_i, i variant de 1 à la taille de la grille) dont les couleurs sont les mêmes que dans A, même si les B_i ne contiennent pas toutes les couleurs de A. Si ce nombre est égal au nombre de couleurs possibles de A, alors nous sommes en présence d'un groupe isolé. Et donc on enlève les couleurs trouvées, dans les cases ne contenant pas le groupe isolé.

2.5 Heuristique du "hidden subset"

Cette heuristique aussi appelée groupes mélangés, obéit au principe suivant : si on ne retrouve N candidats que dans N cases, on est certain qu'ils vont terminer dans ces N cases. Il est donc possible de supprimer les autres candidats de ces N cases, comme on le voit dans l'exemple 2.5.1 avec N=3 :

2.5.1 Exemple

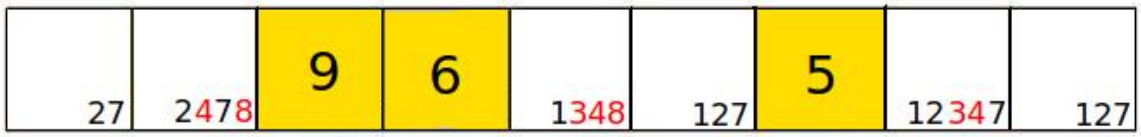

FIG. 2.5.1 – Exemple de hidden-subset

Le groupe mélangé est identifié en rouge, on va donc éliminer les autres candidats de ces trois cases (voir image 2.5.2).

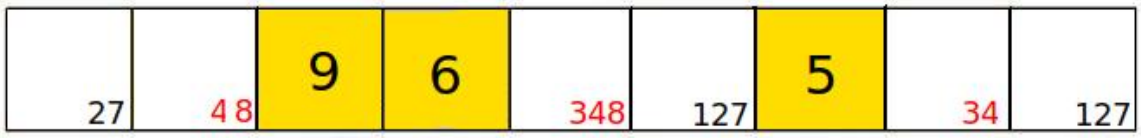

FIG. 2.5.2 – Exemple de hidden-subset

2.5.2 Algorithme du programme

Pour faire fonctionner cette heuristique, on crée un vecteur de pset (appelé i_save dans le programme). Chaque case de ce vecteur contient le pset

représentant l'indice de la case de la sous grille courante dans lequel se trouve chaque couleur (en démarrant les indices de la sous grille courante à 1).
Par exemple, si la couleur 1 se trouve dans les cases 1, 3, 5 et 11 alors la première case du vecteur contiendra un pset qui représentera la chaine de caractère "135B".
Appelons indice_couleur_vecteur, chaque caractère des chaines représentées par les pset du vecteur i_save.
On va appliquer l'algorithme suivant à chaque case du vecteur i_save. Notons i l'indice de la case courante de i_save. On crée un pset (noté uni) qui représente l'union de tous les indice_couleur_vecteurs de la case i et d'une case d'indice j (on réitèrera le processus pour chaque j). Puis on compte le nombre de cases (notons les i_save[l], l variant de 0 à la taille de la grille moins 1) du vecteur i_save dont les indice_couleur_vecteur sont les mêmes que dans uni, sans que ceux de uni soient tous nécessairement dans i_save[l]. A chaque fois que le nombre de cases augmente, on crée un pset (pset_car) qui va représenter la chaine de caractère du hidden-subset et qui est calculé à partir de l. Si on reprend en partie l'exemple précédent, on avait i_save[0]="135B" (si on converti le pset en chaine de caractère). Si on a également i_save[1]="135", i_save[2]="13", et i_save[12]="35B", on aura uni="135B" (union de i_save[0] avec lui-même) et donc i_save[0], i_save[1], i_save[2] et i_save[12] sont inclus dans uni. Et pset_car sera égal à $2^0 + 2^1 + 2^2 + 2^{12}$ et représentera la chaine de caractère "123D" qui sont les couleurs du groupe mélangé potentiel. Si le nombre de indice_couleur_vecteur de uni est égal au nombre de cases qui contiennent un potentiel groupe mélangé alors nous avons trouvé un groupe mélangé. On récupère ensuite chaque indice_couleur_vecteur de uni, car ça va être l'indice des cases (moins 1) où se trouve le groupe mélangé (chaque lettre de A à P codant un chiffre de 9 à 24). Et dans ces cases, on enlève toutes les couleurs qui ne sont pas les couleurs du groupe mélangé.

3 Fonctionnement du solveur

Le solveur est divisé en deux parties, d'abord il applique les heuristiques à chaque sous-grille (colonnes, lignes et régions), puis si les heuristiques ne modifient plus la grille, il utilise le backtracking.

3.1 Heuristiques

Les heuristiques sont appliquées à chaque sous-grille (colonne, ligne ou région). On a donc utiliser un vecteur de pointeurs qui pointent sur les cellules de la sous-grille concernée. Après avoir rempli ce vecteur, on lui applique les

heuristiques. On applique en premier le cross-hatching, pour éliminer un maximum de candidats impossibles dans chaque case. Puis on applique le lone-number. Puis viennent naked-subset et disjoint chain. Bien que naked-subset soit un cas particulier de disjoint-chain, on l'applique avant cette dernière afin d'éliminer les cas les plus évidents de groupes isolés en moins d'efforts. Enfin, on applique l'heuristique nommée hidden-subset. A chaque fois que les heuristiques sont appliqués à une sous-grille, on teste à la fin si la grille a été modifiée. Si la grille n'est plus modifiée, on a alors atteint un point fixe et on arrête les itérations. Si la solution de la grille a été trouvée, c'est-à-dire si toutes les cases n'ont qu'un seul candidat alors on l'affiche et on interrompt le programme. Sinon, on va utiliser le backtracking comme expliqué dans la partie suivante.

3.2 Back-tracking

Le back-tracking est un algorithme qui permet de trouver la solution à coup sûr. Le principe de cet algorithme est de prendre un candidat dans une cellule non encore résolue et de faire la résolution à partir de cette nouvelle grille. Si la résolution mène à une solution, c'est fini, sinon, si on arrive à une nouvelle impasse, on choisit un autre candidat possible dans une autre case et on continue et sinon si on arrive à une grille inconsistante on revient à notre point de départ. Cet algorithme est récursif, on rappelle la fonction qui permet de résoudre la grille.

Dans la pratique, le back-tracking de ce programme fonctionne comme décrit dans la suite. Tout d'abord on sauvegarde la grille afin de pouvoir revenir en arrière si nécessaire. Puis on cherche la case non résolue qui a le plus petit nombre de candidats possibles (l'idéal est 2 candidats encore possible pour cette case). Puis on choisit de mettre dans cette case le premier candidat encore possible. Ensuite, on appelle la fonction qui solve la grille, donc on fait un appel récursif. Dans notre fonction, on teste si la grille est consistante (grâce aux fonctions check_consistency et all_colors) à chaque fois qu'on ré-appelle les heuristiques, si la grille est inconsistante, on revient en arrière en sortant de cet appel récursif. Si la grille est consistante et qu'on a atteint un point fixe, alors on regarde si la grille est résolue ou non. Si elle est résolue, on affiche la grille et on quitte le programme. Si la grille n'est pas résolue, on re-sauvegarde la nouvelle grille, on recherche une nouvelle fois la case avec le plus petit nombre de candidats, et on refait un choix dans cette case. A la sortie de chaque appel récursif, la grille courante est remplacée par la grille précedemment sauvegardée. Puis on teste les autres candidats de la case, parce que le choix du candidat précédent menait à une grille inconsistante. Le back-tracking est résumé dans le diagramme 3.2.1.

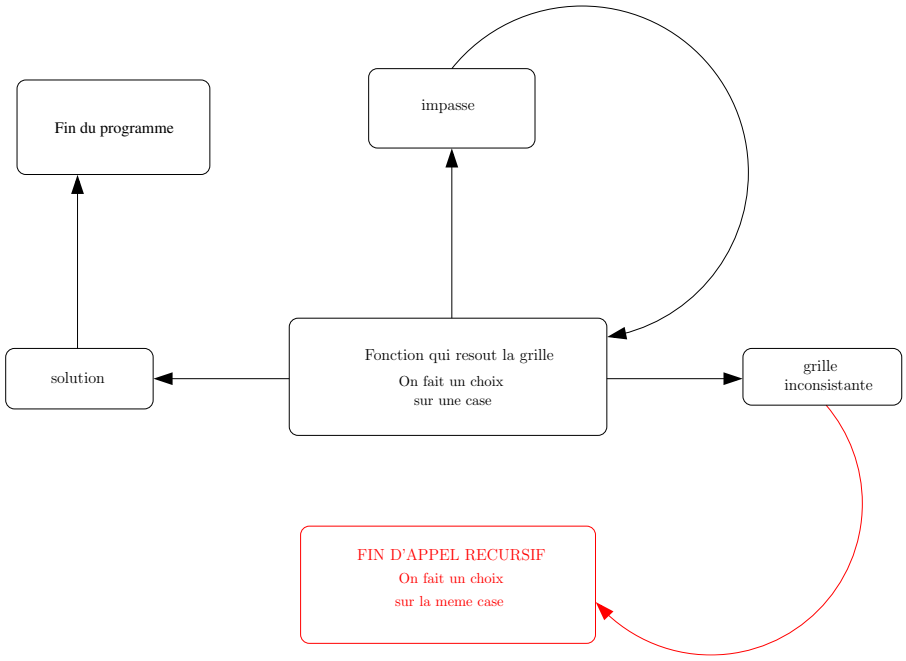

FIG. 3.2.1 – Diagramme représentant le back-tracking servant à la résolution

4 Génération des grilles

Pour générer une grille, il vaut mieux avoir une grille pleine résolue, et ensuite enlever des cases. Les principaux problèmes sont d'avoir des grilles pleines toutes différentes, d'enlever un certain nombre de cases et que les "trous" soient répartis dans toute la grille. L'algorithme de la partie génération du programme est de créer une grille vide, puis de choisir une ligne au hasard, et de remplir chaque case de cette ligne aléatoirement. Ensuite, on résout cette grille grâce à l'algorithme de résolution. On obtient donc une grille pleine qui a été généré aléatoirement. Ensuite, on crée un parcours aléatoire de la grille afin d'enlever des cases. Dans le cas où la grille demandée n'a pas nécessairement qu'une solution, on enlève chaque case du parcours de la grille jusqu'à enlever 6/10 des cases pour une grille de taille inférieure à 25 et 4/10 pour une grille de taille 25 (pour économiser du temps d'exécution). Si la grille générée ne doit avoir qu'une solution, alors à chaque fois qu'on enlève une case, on résoud la nouvelle grille et le solveur nous donne le nombre de solutions. Si la solution n'est pas unique, on remet la case et on poursuit le parcours de la grille. En pratique, le parcours de la grille n'est pas complet, la grille n'est pas entièrement parcourue. Donc si le parcours est terminé et qu'on n'a pas enlevé encore assez de cases, alors on recrée un parcours aléatoire et on recommence jusqu'à enlever un nombre suffisant de cases.

5 Implémentation

5.1 Structures de données

Ce programme utilise principalement le type de données que sont les pset expliqués dans la partie 1.2. Les grilles générées sont des tableaux à deux dimensions de taille donnée par l'utilisateur qui sont de type pset_t**. On utilise dans les heuristiques beaucoup de chaines de caractères afin de compter le nombre de candidats encore possibles dans une case en mesurant la taille des chaines de caractères.

5.2 Découpage du code

En ce qui concerne le découpage du code, le programme est découpé en trois fichiers : le fichier sudoku.c qui est le programme principal et qui contient la fonction main, le fichier preemptive_set.c qui contient toutes les fonctions permettant de manipuler les ensembles préemptifs et le ficher sudoku_generate.c qui s'occupe uniquement de la génération de la grille. Le fait de séparer la génération de la grille avec la résolution me semblait important car certes on a besoin de fonctions de résolution pour générer une grille, mais les deux parties sont bien différentes et le programme est utilisé soit pour résoudre une grille existante soit pour en générer une nouvelle et jamais les deux à la fois.

5.3 Optimisation

L'option -s du programme (lorsqu'elle est utilisée sans -g) est un raccourci pour générer des grilles de taille 9 qui n'ont qu'une seule solution. Ce choix a été fait car les grilles de taille 9 sont les grilles de sudoku les plus souvent utilisées et une vraie grille de sudoku n'a qu'une seule solution.
Les heuristiques utilisant beaucoup de temps d'exécution, elles ont été optimisées. De nombreux tests évitent des tours de boucles inutiles. Dans les heuristiques naked-subset et disjoint-chain, les cases qui sont déjà résolues ne sont pas traitées par le programme car il est inutile de chercher des groupes nus ou des groupes isolés dans les cases qui n'ont plus qu'une couleur possible. Dans l'heuristique hidden-subset, si le nombre de groupes mélangés potentiels est égal au nombre de cases non résolues alors toutes les couleurs non encore fixées sont dans le groupe mélangé et donc on ne va rien changer à la grille. La fonction check_consistency pour tester la consistance de la grille est très longue en temps d'execution. Avant optimisation, elle était utilisée à chaque appel à la fonction subgrid_heuristics dans la fonction grid_solver. Et

dorénavant, elle est appelée seulement une fois avant l'application des heuristiques à toutes les sous-grilles ce qui nous permet de gagner du temps sans perdre l'efficacité du programme.

6 Tests

Le programme passe tous les tests inclus dans le dossier tests, et il génère des grilles de toutes les tailles.

6.1 Utilisation de time

Grace à la commande time, on obtient le tableau 6.1.1. Pour les grilles

	strict	
Taille	Non	Oui
1x1	3 ms	4 ms
4x4	3 ms	4 ms
9x9	20 ms	50 ms
16x16	150 ms	330 ms
25x25	900 ms	1900 ms

FIG. 6.1.1 – Temps de génération des grilles

de taille inférieure à 16, l'utilisateur a l'impression que sa grille s'affiche instantanément, en revanche, pour les grilles de taille supérieure ou égale à 16, il y a un petit temps d'attente. En ce qui concerne la résolution des grilles, la grille grid-25x25-02 du dossier tests est résolue en 800 ms environ. C'est la grille qui met le plus de temps à être résolue et le temps d'attente est acceptable pour l'utilisateur.

6.2 Utilisation de gprof

6.2.1 Résolution des grilles

En compilant avec l'option -pg et en exécutant la grille grid-25x25-02, on obtient, grace à gprof, le tableau 6.2.1 Dans ce tableau on voit que environ un quart du temps est passé dans l'heuristique hidden_subset qui est l'heuristique qui a la plus grande complexité. Puis ce sont les fonctions servant à manipuler les ensembles préemptifs car elles sont appelées de nombreuses

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
27.96	0.26	0.26	6858	0.04	0.04	hidden_subset
22.58	0.47	0.21				pset2str
20.43	0.66	0.19				char2pset
19.35	0.84	0.18				pset_is_included
3.23	0.87	0.03				pset_is_fixed
2.15	0.89	0.02	6858	0.00	0.00	lone_number
1.08	0.90	0.01	6858	0.00	0.00	all_colors
1.08	0.91	0.01	1	10.00	300.00	grid_solver
1.08	0.92	0.01				pset_is_singleton
1.08	0.93	0.01				pset_union
0.00	0.93	0.00	6858	0.00	0.00	cross_hatching
0.00	0.93	0.00	6858	0.00	0.00	disjoint_chain
0.00	0.93	0.00	6858	0.00	0.00	naked_subset
0.00	0.93	0.00	6858	0.00	0.04	subgrid_heuristics
0.00	0.93	0.00	625	0.00	0.00	check_input_char
0.00	0.93	0.00	95	0.00	0.00	check_consistency
0.00	0.93	0.00	1	0.00	0.00	grid_alloc
0.00	0.93	0.00	1	0.00	0.00	print_grid

FIG. 6.2.1 – Tableau obtenu avec grpof suite à l'exécution de la grille grid-25x25-02

fois dans le programme. La fonction check_consistency qui était, avant optimisation, la fonction qui prenait le plus de temps, a maintenant une influence minime sur le temps d'exécution car elle est appelée beaucoup moins souvent.

6.2.2 Génération des grilles

En compilant avec l'option -pg, et en générant une grille de taille 25 avec l'option strict, on obtient avec gprof le tableau 6.2.2. On voit que dans ce cas, 20% du temps est passé dans la fonction pset_is_included qui est appelée principalement dans les fonctions cross_hatching, lone_number et all_colors. On remarque également que la fonction check_consistency prend beaucoup plus de temps que dans le cas de la résolution car elle est appelée environ 16 fois plus à cause du back-tracking. $\sqrt{n}$

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
20.65	0.32	0.32				pset_is_included
19.35	0.62	0.30	123750	0.00	0.00	lone_number
12.26	0.81	0.19	1650	0.12	0.12	check_consistency
10.97	0.98	0.17	123750	0.00	0.00	cross_hatching
9.03	1.12	0.14				pset_is_fixed
7.74	1.24	0.12				pset_is_singleton
7.10	1.35	0.11				char2pset
4.52	1.42	0.07	123750	0.00	0.00	all_colors
3.87	1.48	0.06				pset2str
3.23	1.53	0.05	264	0.19	1.36	grid_solver
0.65	1.54	0.01	123750	0.00	0.00	subgrid_heuristics
0.65	1.55	0.01	1	10.00	440.99	grid_full_generate
0.00	1.55	0.00	25	0.00	0.00	check_input_char
0.00	1.55	0.00	1	0.00	0.00	fill_grid
0.00	1.55	0.00	1	0.00	0.00	grid_alloc
0.00	1.55	0.00	1	0.00	359.01	grid_generate
0.00	1.55	0.00	1	0.00	0.00	print_grid
0.00	1.55	0.00	1	0.00	0.00	random_line

FIG. 6.2.2 – Tableau obtenu avec gprof suite à la génération d'une grille de taille 25 avec une seule solution