Résolution de sudoku

Théo Le Mestre
Maxime Letemple
Mohamed Amine Maalmi
Lucas Pallaro

Objectif

Le programme doit résoudre une grille de sudoku si celle-ci est résoluble.

Modélisation

Nous modélisons la grille de sudoku comme un ensemble de 81 cases. Chaque case est modélisée par sa valeur et par les valeurs qu'elle peut prendre si celle-ci n'en a pas.

Les cases sont liées les unes aux autres car la valeur que peut prendre une case dépend des valeurs présentes sur sa colonne, sur sa ligne et sur son carré. Ces sous-ensembles sont semblables puisqu'ils contiennent chacun 9 cases et obéissent à la même règle : une valeur n'est présente qu'une seule fois dans chaque sous-ensemble. Ainsi des fonctions pourront agir sur un sous-ensemble indifféremment du fait que ce soit une ligne, une colonne ou un carré. Pour une manipulation plus aisée les sous-ensembles seront réunis dans une structure.

Nous résoudrons les grilles par force brute. La méthode est la suivante :
1. Les cases vides qui n'ont qu'une seule valeur possible prennent cette valeur par déduction. On répète cette opération jusqu'à ce qu'il n'y ait plus de case n'ayant qu'une seule valeur possible.
2. On affecte à une case vide une de ses valeurs possibles par supposition et on repart à l'étape 1 de déduction.
Si jamais on arrive à une grille irrésoluble on efface toutes les cases affectées depuis la dernière supposition et on teste une autre supposition. On continue jusqu'à qu'on ce que la grille soit pleine ou qu'on n'ait plus de suppositions possibles.
Un historique sera nécessaire pour savoir quelles suppositions ont été faites et quelles cases ont été affectées pour pouvoir revenir en arrière en cas de mauvaise supposition.

Structure des données

La grille

La grille est représentée par un tableau dont chaque élément est une structure représentant une case.
Structure sudoku_tile représentant une case :
char value
char possible[9]
value contient la valeur de la case et vaut 0 si la case est vide
possible[j] vaut 1 si la valeur j+1 est possible pour value, 0 sinon
Le tableau de la grille contient 81 sudoku_tile et est une variable global ce qui économise un argument pour toutes les fonctions utilisant/manipulant la grille.

Les sous-ensembles

Structure Sudoku_ensemble contenant tous les sous-ensembles de la grille :
sudoku_tile** line[9]
sudoku_tile** col[9]
sudoku_tile** carre[9]

Les champs représentent respectivement les 9 lignes, les 9 colonnes et les 9 carrés. Chaque champ est un tableau de pointeurs. Chacun de ces pointeurs représente lui-même un tableau de pointeurs et ces derniers pointeurs pointent sur les différentes cases de la grille.
Les lignes sont indicées de gauche à droite.
Les colonnes sont indicées de haut en bas.
Les carrés sont indicés dans l'ordre de lecture occidental. Les cases des carrés sont indicées en partant de celle la plus en haut à gauche du carré et dans l'ordre de lecture occidental.

L'historique

L'historique contient 81 éléments. Chaque élément contient les informations suivantes : l'adresse de la case affectée et si cette affectation est le résultat d'une supposition ou bien d'une déduction.
Structure Affectation de chaque élément de l'historique :
sudoku_tile* tile
char supposed
L'historique est rempli dans l'ordre des affectations : le i-ième élément correspond à la i-ième affectation.
L'historique est une variable globale pour un accès simplifié par les fonctions. L'indice de la dernière affectation est également une variable globale.

Initialisation, saisie et affichage de la grille

Ci-après les fonctions d'initialisation, de saisie et d'affichage de la grille et de l'historique.
`void` grid_init (`void`)
Initialise les valeurs des cases à 0 et toutes leurs valeurs possibles à 1
`void` disp_final (`void`)
Affiche la grille
`void` disp_possible (`void`)
Affiche 9 grilles : la grille i montre les cases où la valeur i est possible, impossible et les cases occupées
`void` req_start_grid (`void`)
Permet à l'utilisateur de rentrer une grille via l'entrée standard. Modifie la grille grâce à la fonction suivante
`void` set_tile_value (`sudoku_tile` *tile, `char` value, `char` supposed)
Modifie l'argument tile :
- le champ value de tile prend la valeur de l'argument value
- si l'argument value est différent de 0 (signifiant une case vide) toutes les valeurs possibles de cette case sont mises à 0
En outre la fonction écrit cette affectation dans l'historique. Ecrit une supposition si supposed vaut 1, écrit une déduction sinon
`void` disp_subset (`sudoku_tile**` m_ensemble)
Affiche un sous-ensemble
`Sudoku_ensemble` creation_ensemble (`void`)
Renvoie la structure contenant tous les sous-ensembles de la grille
`void` afficher_history (`void`)
Affiche tous l'historique des affectations

Ci-dessous les résultats :

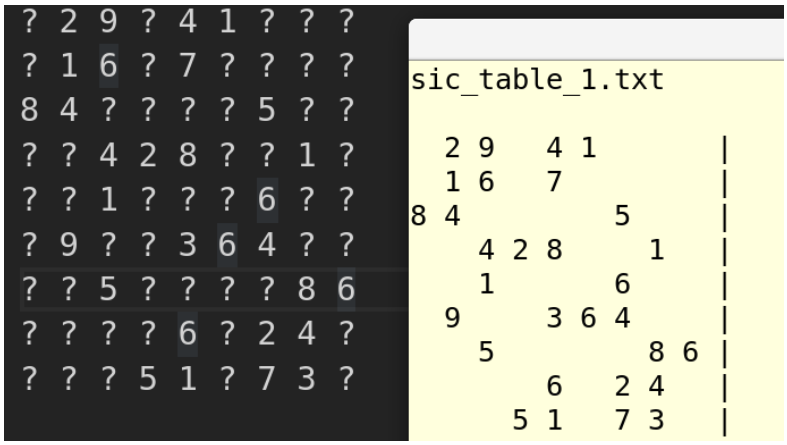

FIGURE 1 : AFFICHAGE D'UNE GRILLE RENTRÉE PAR L'UTILISATEUR (GRILLE RENTRÉE À GAUCHE, TERMINAL À DROITE)

La figure 2 montre le résultat de la fonction disp_possible() : les '+' sont les cases possibles pour la valeur d, les '.' des cases occupées. Les cases occupées de valeur égale à celle de d ont leur valeur affichée.

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

d= 1
+ . . + . 1 + + + |
+ 1 . + . + + + + |
. . + + + + . + + |
+ + . . . + + 1 + |
+ + 1 + + + . + + |
+ . + + . . . + + |
+ + . + + + + . . |
+ + + + . + . . + |
+ + + . 1 + . . + |

d= 2
+ 2 . + . . + + + |
+ . . + . + + + + |
. . + + + + . + + |
+ + . 2 . + + . + |
+ + . + + + . + + |
+ . + + . . . + + |
+ + . + + + + . . |
+ + + + . + 2 . + |
+ + + . . + . . + |
```

FIGURE 2 : AFFICHAGE DE DISP_POSSIBLE()

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

ligne 6
0 9 0 0 3 6 4 0 0
colonne 9
0 0 0 0 0 0 6 0 0
carre 7
0 0 5 0 0 0 0 0 0
```

FIGURE 3 : AFFICHAGE DE DISP_SUBSET()

```
affectation 0 deduction: 5
affectation 1 deduction: 2
affectation 2 deduction: 9
affectation 3 deduction: 3
affectation 4 deduction: 4
affectation 5 deduction: 1
affectation 6 deduction: 6
affectation 7 deduction: 7
affectation 8 deduction: 1
```

FIGURE 4 : AFFICHER_HISTORY(), FORMAT : N°AFFECTATION ET VALEUR AFFECTÉE

Traitement des possibilités uniques

Actualisation des valeurs possibles

Comme la figure 2 le montre un peu plus haut, la valeur d=2 est possible sur des sous-ensembles où se trouve déjà la valeur 2. Les valeurs possibles de chaque case sont actualisées par :
`int` clean_grid (`Sudoku_ensemble` *m_ensemble)
Renvoie 1 si au moins une case a vu une des ses valeurs possibles modifiées. La valeur renvoyée permettra de savoir si on doit sortir ou rester dans une boucle de déduction par exemple.

Déduction dans un sous-ensemble

La déduction à l'intérieur d'un sous-ensemble se fait grâce à la fonction suivante :
`int` valid_exist_in_subset(`sudoku_tile**`m_subset)

Pour chaque valeur v (entre 1 et 9) si une seule case du sous-ensemble contient v comme valeur possible, on affecte v à cette case.
Renvoie 1 si au moins une case est modifiée, 0 sinon.

Boucle de déduction

L'étape de déduction est une boucle qui opère une déduction pour chaque sous-ensemble de la grille. La grille est actualisée entre chaque déduction de sous-ensemble. La boucle se termine lorsque plus aucune déduction n'a lieu dans tous les sous-ensembles et que toutes les cases sont actualisées.

Traitement des suppositions

Affectation de supposition

La fonction suivante nous permet d'effectuer les suppositions :
`int` guess_value (`void`)
Recherche une case pour laquelle la valeur est inconnue et au moins une valeur est possible. Si une telle case existe, la fonction affecte une des valeurs possibles à la case par supposition.
Retourne 1 si une telle case est trouvée, 0 sinon.

Suppositions fausses

Si jamais la supposition est fausse il faut procéder ainsi :
1. Remonter l'historique jusqu'à la dernière supposition grâce à une boucle décrémentant l'indice de l'historique. On affecte la valeur 0 à toutes les cases rencontrées dans cette boucle.
2. Affecter à la dernière case supposée la valeur 0
3. Réinitialiser toutes les valeurs possibles à 1 pour les cases de valeur 0 (une valeur de 0 signifie que la case est vide
4. Utiliser clean_grid() qui permet d'actualiser les valeurs possibles de chaque case de valeur 0 en fonction de la configuration actuelle de la grille.
Si à l'étape 1 la boucle de décrémentation remonte jusqu'à 0 c'est qu'on n'a pas fait de supposition précédemment car toutes ont été essayées. Dans ce cas la grille est irrésoluble.

Ces étapes sont réalisées avec :
`int` back_play (`Sudoku_ensemble` *m_ensemble)
Retourne 1 si une supposition a été supprimée, 0 sinon.

Vérification de la grille

Les deux fonctions suivantes nous permettent de savoir s'il faut poursuivre la résolution/déduction.

`int` is_grid_full (`void`)
Retourne 1 si la grille est complète, 0 sinon.

`int` is_grid_valid (`void`)
Indique si la grille est valide. La grille est invalide si elle contient une case dont la valeur est inconnue, et pour laquelle aucune valeur n'est possible.
Retourne 1 si la grille est valide, 0 sinon.

Réalisation

Algorithme tel que nous l'avons codé en C

Initialisation ...
Tant que la grille n'est pas pleine
    Boucle de déduction
    Tant que des suppositions sont possibles ET que la grille est valide
        Faire une supposition
        Boucle de déduction
    Si la grille n'est pas pleine
        Remonter à la dernière supposition et la marquer impossible
        Si on ne peut pas remonter à la dernière supposition
            Sortir de la boucle

Boucle de déduction : on effectue des déductions en boucle tant que c'est possible.

Pour sortir de la boucle principale il faut soit que la grille soit pleine donc résolue, soit qu'on ait essayé toutes les possibilités.

Conclusion

Avec ceci nous avons pu résoudre toutes les grilles d'exemple avec le programme sudoku_v2.c et résoudre des grilles remplies au hasard.
Remarque : lorsqu'on ne remplit pas suffisamment les grilles le temps de résolution est très long.

Questions pour occuper vos soirées d'hiver

Le type subset est techniquement inutile. Si une fonction a besoin de recevoir un tableau contenant tous les subset de la grille. Quel sera son prototype (sans utiliser le type subset donc) ?

Le prototype sera void fonction (sudoku_tile** argument, [...])

Pourquoi ne pas utiliser directement la solution basée sur les suppositions successives ? quantifiez ... m'enfin, essayez de quantifier :)

Les déductions permettent de réduire le temps d'exécution puisque il est inutile de faire une supposition sur une case s'il n'y a qu'une seule valeur possible.
On peut majorer le nombre de possibilités pour une grille par : 9 puissance nombre_de_cases_vides.
Le nombre de possibilités augmente exponentiellement avec le nombre de cases vides.

C'est un programme où il y a des pointeurs partout ... et peut-être même des mallocs (qui sait). Mais on ne parle jamais d'appels à free, pourquoi ? quand faut-il le faire ? c'est grave si on ne le fait pas ?

Il faut utiliser free() lorsqu'on n'a plus besoin des variables allouées dynamiquement. Or le moment où nous n'en n'avons plus besoin est à la fin du programme donc il est inutile de faire appel à free() : le système d'exploitation se chargera de libérer l'espace alloué. Ainsi ce n'est pas grave dans notre cas bien que ce soit une bonne pratique.