TP #21 — すうどく

Description du problème

Le すうどく (sudoku) est un puzzle consistant à remplir une grille carrée de taille 9×9 par des chiffres entre 1 et 9 de façon à ce qu'aucune ligne, colonne ou sous-carré ne contienne deux fois le même chiffre, où un sous-carré est une sous-grille carrée contiguë de taille 3×3 ancrée en les positions $(0,0)$, $(0,3)$, $(0,6)$, $(3,0)$, $(3,3)$, $(3,6)$, $(6,0)$, $(6,3)$, $(6,6)$.
Une instance d'un problème de sudoku consiste en une grille partiellement remplie ; résoudre un problème consiste à déterminer si cette grille peut être complétée en toutes ses positions non définies de façon à satisfaire les contraintes ci-dessus, et en cas de réponse positive à renvoyer une (ou toutes) ces grilles. Un exemple de grille valide est :

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

L'objectif de ce TP est d'implémenter en C un algorithme de résolution par *backtracking* des problèmes de sudoku.

Représentation des données. On propose de représenter une instance d'un problème de sudoku et son éventuelle solution en la linéarisant par un tableau d'`int` de longueur 81, c'est à dire une valeur de type `int[81]` telle que la $i$ème ligne (en partant de zéro) de la grille de sudoku est stockée dans les éléments du tableau d'indices $\in [\![i \times 9, i \times 9 + 8]\!]$.
Une instance de problème (resp. une solution positive) sera alors un tableau dont tous les éléments sont compris entre `0` et `9` (resp. `1` et `9`) et respectent les contraintes ci-dessus, où `0` représente une case de la grille de sudoku de valeur encore non déterminée.

Fonctions auxiliaires

1. Écrivez une fonction C de signature :
   `void prettyp(int grid[81])`
   qui « affiche joliment » une solution (éventuellement partielle) d'un sudoku. Un exemple de format est :

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

   Testez votre fonction sur les exemples de grilles (correctes ou non) données dans le fichier sud_data.h. Vous devez pour cela au préalable « inclure » le fichier dans le code de votre programme, *via* la directive de préprocesseur :
   `#include "sud_data.h"`
   (en supposant que le fichier `sud_data.h` se trouve dans le même répertoire que votre fichier source).

On va d'abord implémenter un certain nombre de fonctions visant à tester la validité d'une solution (partielle).

2. Écrivez une fonction C de signature :
   `bool row_fine(int g[81], size_t i)`
   qui renvoie `true` si la `i`ème ligne de la grille représentée par `g` contient uniquement des valeurs $\in [\![0,9]\!]$ et qu'aucune valeur $\in [\![1,9]\!]$ n'apparaît plusieurs fois, et `false` sinon.
   On garantira un « coût linéaire en la dimension ligne » pour cette fonction, dans le sens où sa généralisation naturelle à des grilles de dimensions $d \times d$ variables doit avoir un coût en $O(d)$.
3. Écrivez une fonction C de signature :
   `bool rows_fine(int g[81])`
   qui renvoie `true` si toutes les lignes de la grille représentée par `g` sont valides (au sens de la fonction précédente), et `false` sinon.
4. Testez vos fonctions, notamment en utilisant les grilles de test fournies.
5. Écrivez sur le même principe des fonctions C de signatures :

```
bool col_fine(int g[81], size_t j)
bool cols_fine(int g[81])
bool square_fine(int g[81], size_t k)
bool squares_fine(int g[81])
```
qui vérifient si une ou toutes les colonnes ou sous-carrés de g sont valides (au sens d'un sudoku).
Les fonctions vérifiant une unique colonne ou un unique sous-carré devront toujours avoir un « coût linéaire » dans le même sens que précédemment.

6. Testez.
7. Écrivez une fonction C de signature :
   ```
   bool all_fine(int g[81])
   ```
   testant la validité d'une grille de sudoku (éventuellement partielle). C'est à dire que l'on teste simplement si g possède uniquement des lignes, colonnes et sous-carrés valides, et renvoie `true` ou `false` en conséquence.
8. Testez.
9. Écrivez une fonction C de signature :
   ```
   bool set_okay(int g[81], size_t pos, int val)
   ```
   qui pour une grille g **supposée valide** (au sens de la fonction précédente) renvoie `true` si la grille obtenue en y ajoutant un élément de valeur `val` à l'indice `pos` est valide (toujours dans le même sens).
   Cette fonction **ne doit avoir aucun effet observable** (notamment sur g), et doit avoir un coût linéaire (au même sens que précédemment).
10. Testez. (Vous pouvez par exemple notamment vérifier que votre fonction renvoie un résultat correct chaque fois que vous tentez de remplacer à l'identique un élément d'une grille valide).

Trouver une solution

Il est maintenant aisé d'écrire une fonction de résolution de sudoku par backtracking. En effet, étant donnée une grille partielle valide, il suffit de parcourir toutes les cases non déterminées (dans un ordre quelconque, mais l'ordre croissant des indices est le plus naturel) et pour chaque case, échouer si elle ne peut prendre aucune valeur $\in [\![1, 9]\!]$ et sinon résoudre récursivement toutes (si besoin) les grilles obtenues en lui affectant les valeurs qui sont *okay* ; on s'arrête avec succès quand toutes les cases sont déterminées.

11. Écrivez une fonction C récursive de signature :
    ```
    bool _solve(int g[81], size_t pos)
    ```
    qui résout l'instance de sudoku décrite par g supposée *fine* et dont toutes les cases d'indice strictement inférieur à `pos` sont déterminées (ont une valeur $\in [\![1, 9]\!]$).
    Cette fonction doit renvoyer `true` si l'instance décrite par g possède une solution et la modifier en une solution, et sinon renvoyer `false` **et laisser** g

inchangée. Cette dernière contrainte est cruciale pour le bon fonctionnement du backtracking.
N.B.. Cette fonction n'a aucune raison d'être très longue ; on s'en sort très bien en une vingtaine de lignes dont la moitié sont quasiment vides.

12. Prouvez sa terminaison.

13. Déduisez-en une fonction C de signature :
```
bool solve(int g[81])
```
qui résout l'instance de sudoku décrite par `g` en renvoyant `true` si elle possède une solution et la modifie en une solution, et `false` sinon et la laisse inchangée.

14. Testez, notamment en utilisant les grilles de test fournies.

Afficher toutes les solutions

15. Écrivez une variante de votre fonction `_solve` de signature :
```
void _solve_print_all(int g[81], size_t pos)
```
qui pour une instance de sudoku décrite par `g` supposée *fine* et dont toutes les cases d'indice strictement inférieur à `pos` sont déterminées en affiche toutes les solutions possibles.
N.B. Votre fonction ne doit jamais afficher plusieurs fois la même solution, mais il n'y a pas de contrainte sur l'ordre dans lequel vous énumérez celles-ci. Cependant, le plus simple est sans doute de le faire dans l'ordre lexicographique ; autrement dit, d'énumérer les valeurs possibles en `g[pos]` dans l'ordre naturel des entiers.

16. Écrivez de même une variante de `solve` de signature :
```
void solve_print_all(int g[81])
```

17. Testez. Vous devriez notamment trouver une et deux solutions respectivement pour les grilles d'exemple `g1` et `g2`.

Générer toutes les solutions

Une seule instance d'un problème de sudoku peut avoir *beaucoup* de solutions (c'est le cas par exemple de la grille sans aucune contraintes), et un algorithme affichant (ou renvoyant sous un format quelconque, cela revient au même) *toutes* les solutions peut donc avoir un coût prohibitif simplement à cause de leur grand nombre.
On peut bien sûr limiter le nombre de solutions calculées lors d'une résolution, mais cela est un peu restrictif et pas forcément satisfaisant ; on peut par exemple imaginer un contexte d'accord, dans le cas du sudoku c'est sans doute difficile dans lequel le nombre de solutions désirées est inconnu *a priori*, et l'on ne veut ni trop en générer (c'est inefficace) ni pas assez (ça l'est aussi, s'il faut recommencer la recherche depuis le début pour en générer de nouvelles).

Il peut donc être intéressant d'écrire une fonction (en fait, deux) qui permettent de *générer* (jusqu'à) toutes les solutions d'une instance *à la demande* de l'utilisateur ou utilisatrice.
Pour cela on propose de suivre la stratégie suivante : on définit le *contexte* d'une solution (partielle) par un type :

```
struct sol_ctx
{
    int g[81];
    struct stack mods;
    size_t pos;
};
```

où le champ `g` représente une grille (éventuellement incomplète), le champ `pos` le plus petit `i` tel que `g[i]` vaut `0` (la grille est indéterminée en cette position) s'il y en a un, et `mods` une pile des indices *inférieurs ou égaux* à `pos` où `g` a été modifiée par rapport à la grille définissant l'instance du problème.
Alors, produire une (nouvelle) solution (éventuellement partielle) à partir d'un tel contexte consiste à chercher une valeur possible pour `g[pos]` (parmi celles qui n'ont pas déjà été testées) puis renvoyer la solution obtenue si la grille est complète ou s'appeler récursivement sur un nouveau contexte reflétant le choix effectué, ou s'il n'y en a pas s'appeler récursivement sur l'indice en sommet de pile ou échouer si cette dernière est vide.

18. Définissez un type `struct stack` de pile impérative permettant de stocker des valeurs de type `size_t`, ainsi que des fonctions de manipulation de signatures :

```
bool is_empty(struct stack *s)
void push(struct stack *s, size_t v)
size_t pop(struct stack *s)
```

On conseille *très vivement* de faire au plus simple, et notamment de tirer profit du fait que cette pile n'est utilisée que dans des contextes (haha) où son nombre d'éléments est majoré par 81.

19. Écrivez une fonction C récursive de signature :

```
struct sol *solve_next(struct sol_ctx *s)
```

qui implémente l'approche esquissée ci-dessus. (Un peu) plus précisément, `solve_next(s)` doit renvoyer une valeur de pointeur nul s'il n'existe pas de nouvelle solution (pas déjà renvoyée) que l'on puisse obtenir à partir du contexte `*s`, et sinon un (pointeur vers un) contexte contenant une nouvelle solution ainsi que les informations permettant de poursuivre plus tard la recherche d'éventuelles nouvelles solutions.
On conseille d'écrire la fonction de façon à ce que sa première étape consiste à dépiler `s->mods` et de poursuivre la recherche à partir de la position ainsi dépilée ; le cas où la pile serait vide est alors l'unique point de la fonction où

une valeur de pointeur nulle doit être renvoyée.
N.B. Les contraintes sur l'ordre d'énumération sont les mêmes que pour `_solve_print_all`.

20. Écrivez une fonction C de signature :
```
struct sol *solve_gen(int g[81])
```
qui renvoie un pointeur nul si g n'est pas *fine*, et sinon un contexte représentant sa première solution (pour votre ordre d'énumération) ainsi que les informations nécessaires pour énumérer les suivantes avec `solve_next`.

21. Testez.

Remarques. On peut observer une certaine similarité entre la représentation du contexte utilisée ici et une structure de donnée de tableau persistant (en fait *semi-persistant*) : bien que la grille soit modifiée lors de la recherche de solution, on doit pouvoir revenir à un état précédent.

L'approche suivie ici peut-être adaptée dans un style fonctionnel en utilisant une *continuation* pour stocker le contexte ; ceci donne une approche très générale pour implémenter une *inversion de contrôle* comme effectuée ici.

Dérécursification

On peut (normalement) constater que contrairement à la fonction `_solve`, `solve_next` est récursive terminale ce qui n'est pas vraiment un hasard. On peut donc facilement réécrire cette dernière en style itératif.

22. Écrivez une fonction C non récursive de signature :
```
struct sol *solve_next_iter(struct sol_ctx *s)
```
de mêmes spécifications que `solve_next`.

23. Testez.