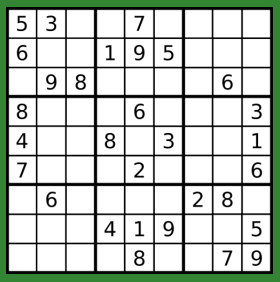
UGMT41 — Algorithmique et Projet de programmation

Master 1 de Mathématiques et applications
Université Paris Diderot



Projet de programmation en C : solveur de sudoku

Pierre Cagne

pierre.cagne@normalesup.org

Résumé

Ce projet de programmation a pour but de vous faire coder un solveur du jeu de sudoku par des méthodes avancées de *backtracking*. Ce domaine de l'algorithmique est dévoué à l'étude des méthodes de recherche exhaustive de solutions à un problème donné en évitant tant que possible la méthode dite de *brute-force* (qui consiste à essayer toutes les possibilités pour en extraire les solutions).

À rendre pour le 06 janvier 2017

L'implémentation d'un solveur de sudoku peut prendre différentes formes. L'idée la plus simple serait de tester toutes les combinaisons possibles de remplissage de la grille et de ne garder que celle(s) satisfaisant aux règles classiques du sudoku : un unique i par ligne, colonne et bloc pour $i \in \{1\,,\ldots,9\}$. C'est ce que l'on appelle la méthode *brute-force* : elle teste les m^9 remplissages possibles, m étant le nombre de cases vides de la grille (typiquement m est de l'ordre de 60 à 65), y compris les remplissages les plus stupides, avec par exemple deux 1 l'un à côté de l'autre... Autant dire qu'il y a de la place pour améliorer cela !

Le but de ce projet est de vous faire implémenter un solveur plus efficace grâce à un processus en trois étapes[1] : une modélisation mathématique abstraite du problème à résoudre ; un traitement algorithmique du problème abstrait ; une implémentation efficace de l'algorithme, grâce aux outils du langage à disposition. Bien entendu, on n'attend pas de vous que vous fassiez tout cela tout seul, et les trois premières parties de ce document sont justement là pour vous guider.

La première étape, en section 1, peut sembler à première vue complexifier le problème. En fait, en incluant le sudoku dans une classe plus large de « puzzles », on identifie précisément ce qui fait la complexité de la résolution d'une grille de sudoku. La deuxième étape, présentée en section 2, va permettre d'encoder les notions abstraites de la première étape comme des objets concrets, afin de concevoir un algorithme efficace résolvant le problème. Enfin, la troisième étape, en section 3, vous invite à utliser tous les outils mis à votre disposition par le langage `C` pour coder effectivement les objets et l'algorithme pensés à la deuxième étape. La section 4 détaille les modalités de rendu du projet.

Remarque. Les trois premières parties doivent se lire avec un papier et un stylo à la main, et **surtout pas** avec un clavier et un fichier `.c` ouvert à l'écran...

Dans tout le document, on fait l'hypothèse que l'on ne manipule que des ensembles *finis*.

1 Problème de satisfaction de contraintes

Supposons qu'on ait deux ensembles P et C et une relation $\mathcal{S} \subseteq P \times C$. On appelera les éléments de P des *possibilités* et ceux de C des *contraintes*. Comme d'habitude, on notera $p\ \mathcal{S}\ c$ plutôt que $(p\,,c) \in \mathcal{S}$ pour des éléments $p \in P$ et $c \in C$, et on dira dans ce cas que p *satisfait la contrainte* c.

Définition. Un tel triplet $(P\,,C\,,\mathcal{S})$ est appelé *problème de satisfaction de contraintes*.

Une *solution* du problème $(P\,,C\,,\mathcal{S})$ est un sous-ensemble $P^* \subseteq P$ tel que toute contrainte $c \in C$ soit satisfaite par **exactement une** possibilité $p \in P^*$. Autrement dit, c'est une fonction $f : C \to P$ telle que $f(c)\ \mathcal{S}\ c$ pour tout $c \in C$.

Remarque. Attention, le vocabulaire utilisé dans ce document est hautement **non standard**. En particulier, googliser « problème de satisfaction de contraintes » ou « constraint satisfaction problem » risque de vous emmener vers des considérations mathématiques et algorithmiques qui n'ont que très peu à voir avec les notions présentées ici.

1. C'est un processus que vous pouvez appliquer à la plupart des problèmes concrets que vous souhaitez résoudre informatiquement.

Malgré son apparente abstraction, la notion de problème de satisfaction de contraintes est omniprésente, en mathématiques comme dans la vie courante. En voici quelques exemples.

Exemples.

(1) Soit X un ensemble. Les solutions au problème de satisfaction de contraintes $(\mathcal{P}(X)\,,X\,,\ni)$ sont exactement les partitions de X.

(2) On trouve souvent des problèmes de satisfaction de contraintes dans les revues de divertissement sous forme de « puzzles logiques ». Ce sont ces problèmes logiques posés sous la forme suivante :

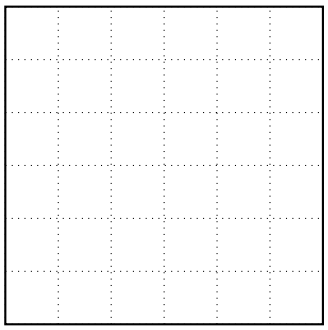
> Alice, Bob et Dave ont fait des emplettes. L'un a acheté des carottes, un autre des pommes de terres et le dernier des oignons. Ils ont fait leurs achats chacun différemment : en grande surface, au marché ou chez l'épicier. Retrouver qui à acheter quoi et où, sachant que :
>
> (C_1) Alice n'aime pas la grande distribution,
>
> (C_2) L'épicier n'avait plus de carottes,
>
> (C_3) Les pommes de terre ont été achetés au marché,
>
> (C_4) L'un de Bob ou Dave est allé chez l'épicier,
>
> (C_5) Dave déteste les oignons.

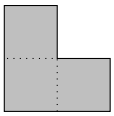
On peut modéliser le puzzle comme un problème de satisfaction de contraintes : l'ensemble P des possibilités sont les triplets $(\text{nom}\,,\text{produit}\,,\text{lieu})$ et l'ensemble C des contraintes est donné par $\{C_1\,,C_2\,,C_3\,,C_4\,,C_5\}$ auquel on ajoute les contraintes implicites du problème : une contrainte par nom (notons les A pour Alice, B pour Bob et D pour Dave), une par produit (c pour carottes, p pour pommes de terre, et o pour oignons), et une par lieu (e pour épicier, m pour marché et s pour supermarché). Bien entendu, la relation $\mathcal{S}$ de satisfaction est définie comme la satisfaction de la contrainte au sens courant du terme : par exemple,

$$\begin{aligned}(\text{Alice}, \text{oignons}, \text{supermarché})\ &\mathcal{S}\ A,\\ (\text{Alice}, \text{carottes}, \text{supermarché})\ &\not\mathcal{S}\ C_1,\\ (\text{Bob}, \text{pommes de terre}, \text{épicier})\ &\mathcal{S}\ C_4, \quad \text{etc.}\end{aligned}$$

Dans le cas présent (et dans la plupart des puzzles de ce genre), il y a une unique solution au problème de satisfaction de contraintes.

(3) Supposons qu'un maçon veuille paver la cour d'un jardin (représentée à gauche) avec des tuiles toutes de la même forme (représentée à droite) :





Il a le droit de placer les tuiles dans le sens qu'il veut et où il veut dans la cour, mais à la fin les tuiles ne doivent pas se superposer et chaque carré de la cour doit être couvert.

Pour énumérer toutes les possibilités et choisir parmi celles-ci, il peut modéliser la situation comme un problème de satisfaction de contraintes comme suit : l'ensemble P des possibilités est l'ensemble des triplets $(x_1\,, x_2\,, x_3)$ avec $x_1\,, x_2\,, x_3 \in [1\,, 6] \times [1\,, 6]$ d'une des formes suivantes :

$$\begin{array}{ll} ((i, j+1), (i, j), (i+1, j)) & ((i, j-1), (i, j), (i+1, j)) \\ ((i, j-1), (i, j), (i-1, j)) & ((i, j+1), (i, j), (i-1, j)) \end{array}$$

C'est-à-dire que P est l'ensemble des placements possibles de la tuile (rotations comprises) dans la cour. L'ensemble C des contraintes est $[1\,, 6] \times [1\,, 6]$, c'est-à-dire l'ensemble des carrés de la cour. La relation de satisfaction, définie par

$$(x_1, x_2, x_3)\ \mathcal{S}\ c \iff \exists i,\ x_i = c,$$

exprime alors simplement le fait qu'un carré c est recouvert par le placement en $(x_1\,, x_2\,, x_3)$ de la tuile. Une solution au problème $(P\,, C\,, \mathcal{S})$ est donc la sélection P^* d'un certain nombres de placements de tuiles de telle sorte que chaque carré de la cour soit recouvert par exactement une tuile de P^* : c'est bien la définition d'un pavage de la cour. Par exemple, on a la solution suivante (bien entendu, ce n'est pas la seule) représentée graphiquement en figure 1 :

$$P^* = \left\{ \begin{array}{l} ((1,2),(1,1),(2,1)),\ ((4,2),(4,1),(5,1)),\ ((1,4),(1,3),(2,3)), \\ ((4,4),(4,3),(5,3)),\ ((1,6),(1,5),(2,5)),\ ((4,6),(4,5),(5,5)), \\ ((2,2),(3,2),(3,1)),\ ((5,2),(6,2),(6,1)),\ ((2,4),(3,4),(3,3)), \\ ((5,4),(6,4),(6,3)),\ ((2,6),(3,6),(3,5)),\ ((5,6),(6,6),(6,5)) \end{array} \right\}$$

Question 1. Montrer que le jeu de sudoku est un problème de satisfaction de contraintes. Plus exactement, si G est une grille de sudoku, contruire un problème $(P_G\,, C_G\,, \mathcal{S}_G)$ de telle façon que les solutions P_G^* soient (en bijection avec) les remplissages valides de la grille G.

2 Résolution algorithmique

La question suivante permet de reformuler les problèmes de satisfaction de contraintes dans un langage plus adapté à l'implémentation en C.

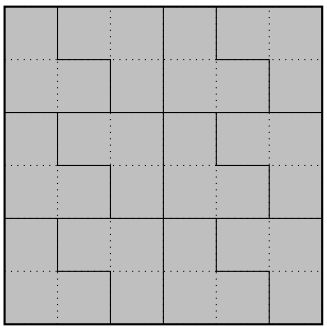


FIG. 1: Exemple de pavage

Question 2. Soit $(P, C, \mathcal{S})$ un problème de satisfaction de contraintes. Constuire une matrice $M = (m_{i,j})_{1 \leq i \leq p, 1 \leq j \leq q}$ (à vous de trouver p et q) à coefficients $m_{i,j} \in \{0, 1\}$ de telle façon que les solutions du problème $(P, C, \mathcal{S})$ soient en bijection avec l'ensemble

$$\left\{ (i_1, i_2, \ldots, i_n) \middle| \begin{pmatrix} m_{i_1,1} & m_{i_1,2} & \ldots & m_{i_1,q} \\ m_{i_2,1} & m_{i_2,2} & \ldots & m_{i_2,q} \\ \vdots & \vdots & \ddots & \vdots \\ m_{i_n,1} & m_{i_n,2} & \ldots & m_{i_n,q} \end{pmatrix} \text{ a exactement un } 1 \text{ par colonne} \right\}$$

À partir de maintenant, on appelle donc aussi *solution* du problème un tel uplet $(i_1, \ldots, i_n)$, et on dira que la matrice M *présente* le problème $(P, C, \mathcal{S})$.

La question de trouver toutes les solutions d'un problème de satisfaction de contraintes est donc maintenant ramenée à l'élaboration d'un algorithme déterminant toutes les sélections de lignes possibles d'une matrice à coefficients booléens (i.e. dans $\{0, 1\}$) de telle manière que la matrice résultante ait un unique 1 dans chaque colonne.

Question 3. Remarquer qu'une solution contenant une ligne fixée est exactement la même chose qu'une solution d'un problème plus restreint. Plus précisément, si on fixe une ligne i, une solution $(i_1, \ldots, i_n)$ du problème présenté par M pour laquelle $i_k = i$ est exactement la donnée d'une solution $(i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_n)$ pour le problème présenté par la matrice obtenue à partir de M comme suit :

- on enlève toute les lignes de M qui ont un 1 dans toutes les colonnes où la ligne i en a un,
- on enlève également les colonnes ayant un 1 en ligne i.

À partir de cette observation, déterminer un algorithme récursif dont l'entrée est une matrice à coefficients booléens et dont la sortie est l'ensemble des solutions au problème présenté par cette matrice.

3 Implémentation efficace

On sait représenter en `C` une matrice par un tableau de tableaux. On pourrait donc croire qu'on a tous les outils en main pour coder effectivement l'algorithme.

Même si cela est vrai dans l'absolu, on risque d'être confronté à un problème d'ordre pratique : les matrices que l'on manipule sont de grandes tailles, et elle contiennent étonnement peu de 1. L'algorithme élaboré précédemment risque donc de passer énormément de temps à chercher des 1 dans des colonnes remplies de 0...

Une idée pour contourner le problème est de ne garder la représentation matricielle que virtuellement (dans l'esprit du programmeur) et de n'avoir en mémoire (celle de l'ordinateur) que la position relative des 1 les uns par rapport aux autres.

Question 4. Créer une structure `C` :

```
typedef struct node_s node_t;
struct node_s {
  node_t *left, *right, *up, *down;
  /* other fields as you may wish */
};
```

Chaque noeud représentera un 1 de la matrice booléenne, et devra pointer vers 4 autres noeuds représentant ses prédécesseurs et successeurs dans sa ligne et sa colonne. Si un tel successeur (respectivement prédécesseur) n'existe pas, le pointeur ira vers le premier (respectivement dernier) 1 de la ligne ou colonne en question.

Il vous faudra également créer une structure :

```
typedef struct column_s column_t;
struct column_s {
  node_t head;
  column_t *left, *right;
  /* your call for other fields */
};
```

représentant la « tête » des colonnes de la matrices. On réservera un instance particulière de cette structure pour marquer le point d'entrée dans la matrice.

On pourra s'aider du dessin de la figure 2, où chacun des carrés marqués 1 est un `node_t` et chaque carré de la première ligne est un `column_t`, et qui représente la matrice suivante :

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Cette manière de représenter les matrices booléennes en mémoire a aussi l'avantage de pouvoir effectuer *en place* la sélection de la sous-matrice expliquée en question 3.

Question 5. Que fait la fonction suivante ?

```
void mystery (node_t* x) {
  x->left->right = x->right;
  x->right->left = x->left;
}
```

En déduire une implémentation de l'algorithme de la question 3 qui ne crée qu'une seule instance de matrice en mémoire.

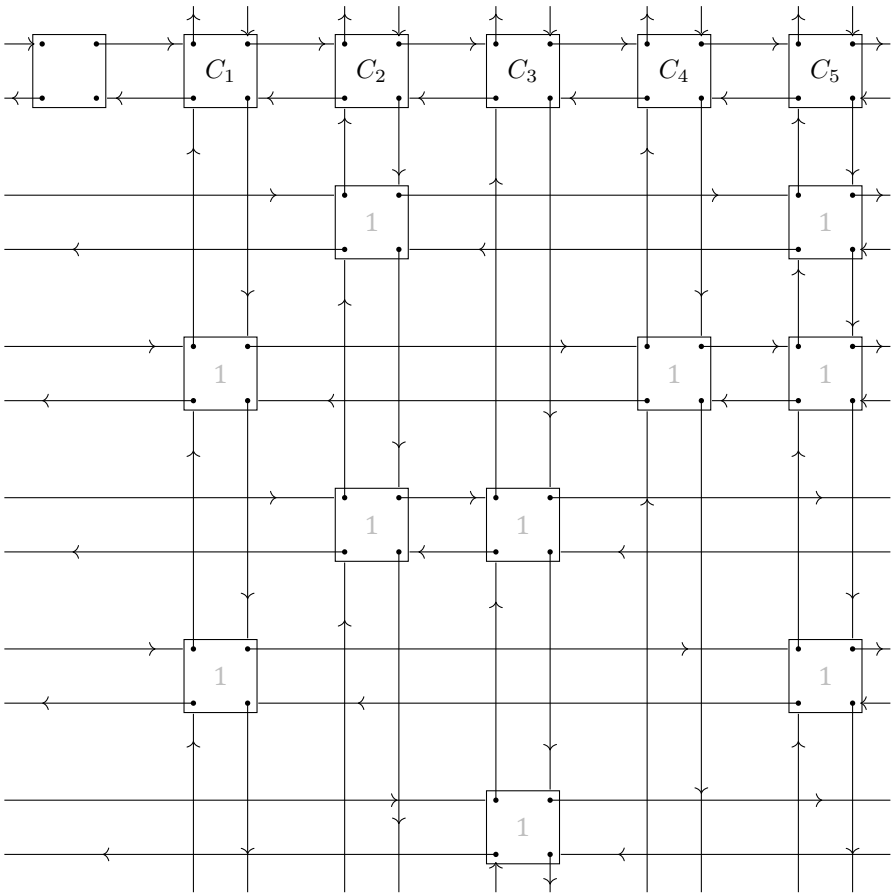


FIG. 2: Représentation graphique des structures `node_t` et `column_t`.

4 À rendre

Toutes les questions préliminaires sont à faire sérieusement, sur une feuille de papier, pour votre propre compréhension du sujet. En revanche, elles ne sont pas à rendre formellement comme partie intégrante du projet.

Ce qu'il faut rendre, c'est un ensemble de fichiers `.h` et `.c`, compressés en une archive `.zip` nommée `UGMT41_nomdefamille.zip`. Le fichier `.c` contenant la fonction `main` devra s'appeler `main.c`. En particulier, la compilation avec la ligne de commande :

```
gcc -Wall -Wextra -o sudoku *.c
```

ne devra retourner aucune erreur (le mieux étant qu'elle ne retourne aucun warning non plus).

On dira qu'un fichier texte est au *format sudoku* s'il se compose d'exactement 9 lignes, chacune de ces lignes étant composée d'une chaîne de 9 caractères, chacun de ces caractères pouvant être `'0'`, `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, `'6'`, `'7'`,`'8'` ou `'9'`. Informellement, le caractère `'0'` représente une case vide de la grille de sudoku à

résoudre, tandis qu'un des autres caractères représente une case préremplie de la grille. Ainsi, le fichier contenant :

```
530070000
600195000
098000060
800060003
400803001
700020006
060000280
000419005
000080079
```

représente la grille de sudoku de la figure 3.

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

FIG. 3: Exemple de grille de sudoku

Le programme devra accepter un paramètre passé en ligne de commande :

```
./sudoku path_to_sudoku_file
```

où `path_to_sudoku_file` est le chemin d'accès à un fichier texte au format sudoku (au sens expliqué ci-dessus). L'éxécution devra alors renvoyer à l'écran la liste des solutions. Si plusieurs solutions existent, elles devront être affichées, chacune au format sudoku, avec une ligne vide les séparant.

Remarque. **Attention** de bien respecter ces conventions, une partie de la correction étant automatisée : un petit programme de mon cru testera votre programme sur un grand nombre de sudokus et comparera le résultat à un fichier test contenant les solutions. Tout manquement aux conventions peut donc se solder à un échec du test même si votre algorithme est juste en soi !

L'implémentation de l'algorithme élaboré dans les parties 1 à 3 et le respect des règles décrites ci-dessus est le *strict minimum*. C'est-à-dire que tout manquement aux prérogatives précédentes se ressentira drastiquement dans la note. En particulier, on insiste sur le fait que le projet rendu doit suivre l'algorithme décrit précédemment ! Bien entendu, de petites améliorations peuvent être apportées ici et là, mais le coeur de l'algorithme doit rester celui développé en partie 2. Par

exemple, un projet résolvant la grille de sudoku par *brute-force* sera automatiquement écarté.

Ceci étant dit, le passage des prérogatives de cette seciton n'est pas synonyme d'une note fixée : le code lui-même sera partie intégrante de l'évaluation (un code concis, clair et élégant sera toujours préféré à un code brouillon et peu lisible même si la sémantique des programmes résultants est la même). **Dans tous les cas, un code non commenté sera grandement pénalisé.**

Enfin, toute amélioration du programme sera vivement appréciée et pourra se voir attribuer des *points bonus*. Voici une liste non-exhaustive de personnalisations possibles :

- proposer à l'utilisateur de rentrer la grille de sudoku à résoudre lignes à lignes en mode intéractif,
- accepter un format plus large de grille de sudoku en entrée,
- créer automatiquement des grilles de sudoku certifiées valides (i.e. dont on est sûr, grâce au solveur, qu'elles admettent une unique solution),
- proposer de résoudre d'autres problèmes de satifaction de contraintes que le sudoku,
- proposer un mode *verbose* qui montre toutes les étapes de résolution (attention, cela peut vite rendre la sortie indigeste pour l'utilisateur),
- proposer **en plus** un mode *brute-force* permettant de comparer le temps de calcul avec la méthode implémentée,
- etc.

Pour être évaluée, toute amélioration devra venir avec un documentation (interne ou externe au programme). Elle devra aussi être compatible à la *version standard* : c'est-à-dire que si l'on fait abstraction des améliorations, le programme doit fonctionner **exactement** comme décrit dans les prérogatives à remplir. Un manquement à cette dernière règle se verrait très pénalisé, puisque le *strict minimum* ne serait alors pas atteint.

Le projet est à rendre pour le 14 janvier 2017. Vous êtes bien entendu invités à collaborer à plusieurs dans votre réflexion, comme sur n'importe quel devoir maison. En revanche, le projet final doit être fait seul ou en binôme. Nous recommandons vivement l'option binôme : la collaboration (souvent à bien plus que deux) est souvent de mise dans les projets informatiques *de la vie réelle*. On conseille alors d'utiliser un logiciel de *subversionning* pour vous simplifier la tâche : les plus courant sont `git`, `svn` et `mercure`.