# Low Level Programming
# Project

Briac Bruneau

**Résumé**

Ceci est un résumé du document.

## Table des matières

**1 Structures** **1**

**2 Résolution** **1**
2.1 Heuristiques . . . . . 2
2.1.1 Cross-hatching . . . . . 2
2.1.2 Lone number . . . . . 2
2.1.3 Naked-subset . . . . . 2
2.1.4 Hidden-subset . . . . . 2
2.2 Choix et Backtrack . . . . . 3

**3 Génération** **3**

**4 Optimisation** **4**

## 1 Structures

On a choisi de représenter chaque case de la grille comme un entier allant de 0 a $2^n$ en binaire, n étant la taille de la grille donc un des entiers parmi {1,4,9,16,25,36,49,64}. Chaque composante du binaire déterminera si le chiffre qui lui correspond peut être placé dans cette case ou non, par exemple, sur une grille de taille 4, si un case est représentée comme : 1010 en base 2, les chiffres 4 et 2 sont possibles sur cette case, mais pas 1 et 3. On appelera ce binaire une couleur.

La grille sera ensuite représentée par un pointeur de pointeurs de " couleurs ", ce qui nous permettra d'effectuer des opérations sur la grille avec le plus d'efficacité possible par le biais d'opérations sur les bits des couleurs. On choisit d'imbriquer le module couleur dans le module grille qui est lui même dans le module sudoku qui sera notre main, permettant ainsi de s'assurer qu'on ne puisse pas toucher aux cases depuis nos fonctions principales.

## 2 Résolution

Le programme contient deux composantes majeures, la résolution et la génération de sudoku. On donne un sudoku en entré en appelant le programme, et il nous renvoie une solution ou bien toutes les solutions possibles si on utilise -a ou –all en argument. La difficulté de résolution des sudoku peut beaucoup varier, certains prenant des millièmes de secondes quand a d'autre prenant au programme plusieurs minutes.

On a implémenté 4 heuristiques qui nous serviront pour résoudre le sudoku, il en existe d'autres qu'il serait possible d'implémenter afin d'optimiser le programme ainsi que de déterminer des conditions à leur utilisations, certains étant plus couteux que d'autres et donc moins efficaces a appeler a chaque boucle.

On divise la grille en 3 types de sous-grilles : les lignes, les colonnes et les carrés. On utilise ensuite nos heuristiques sur chacune de ces sous-grilles a chaque fois, commencant par les plus efficaces qui servent a faire le gros du boulot dans la majorité des cas : cross-hatching et lone number. On utilise ensuite deux autres heuristique une fois que celles-ci ne changent plus rien dans notre grille : naked-subset et hidden-subset qui ont un coût plus conséquent.

## 2.1 Heuristiques

### 2.1.1 Cross-hatching

L'heuristique Cross-hatching est celle que l'on utilise le plus pour la résolution d'un sudoku "simple", je l'ai implémentée en $O(n^2)$. On parcourt toute la sous-grille et a chaque fois qu'on trouve un singleton, on l'enlevait dans les autres cases de la sous-grille. ( un singleton étant une couleur avec un seul bit égal à 1 donc on est certain que la case contient le chiffre correspondant à ce bit ). Cette heuristique est très utile en début de résolution quand on a encore beaucoup de cases vides qui correspondent a " tous les chiffres sont possibles ", elle élimine donc beaucoup de possibilités d'un coup. Puis a chaque fois qu'on trouve une case avec une autre heuristique, elle nous est a nouveau utile pour éliminer ce chiffres pour les autres cases des sous-grilles dont la première case fait partie.

### 2.1.2 Lone number

L'heuristique Lone number permet de trouver quand une case dans une sous-grille contient un chiffre isolé : cette case contient alors forcément le chiffre isolé sinon la grille est inconsistante. Pour faire cela on utilise des bit-hacks : on crée deux binaires, un va servir a contenir les chiffres qui sont en plusieurs fois et un autre pour ceux qui y sont au moins une fois. On compare a chaque fois la case avec les chiffres qui sont déja dans la sous-grille au moins une fois et on ajoute ceux en commun au binaire contenant les chiffres compris plusieurs fois dans la sous-grille. A la fin de la boucle, tout ceux qui ne sont pas dans le binaire contenant les chiffres qui sont en plusieurs fois dans la sous-grilles n'y sont qu'une seule fois et il nous suffit d'un deuxieme tour parcourant la sous-grille pour trouver a quel case ils appartiennent. Il est possible que 2 de ces chiffres appartienent à la même case, dans ce cas la grille est inconsistente. Cette heuristique est implémentée en $O(n)$.

### 2.1.3 Naked-subset

Les deux heuristiques suivant font intervenir des subsets correspondant a un sets de plusieurs chiffres présents dans plusieurs cases. Avec le naked-subset, on regarde si un sous-set de N candidats apparait dans N cases en $O(n^2)$, dans ce cas tout les chiffres de ce sous-set peuvent être éliminés des autres cases.

Je choisis de parcourir la sous-grille, à chaque case on crée un sous-set égal a tous les candidats de la case, si cette case n'est pas un singleton on parcourt les autres cases en incrémentant un compteur si celle-ci est contenue dans le sous-set. si le compteur est égal au nombre de candidats du sous-set on peut procéder à la suite. on retire tous ses candidats aux autres cases ( on vérifie que ce sont d'autres cases en comparant leur intersection avec elles-même ).

### 2.1.4 Hidden-subset

Pour la dernière heuristiques hidden-susbet, on utilise également un sous-set que l'on compare avec les autres cases de la sous-grille, mais cette fois on regarde si la case contient le sous-set et si on a autant de cases validant la condition que de chiffres dans le sous-set on peut enlever tous les autres chiffres des cases candidates n'appartenant pas au sous-set, cette heuristique est également implémentée en $O(n^2)$.

L'implémentation de cette heuristique est la plus compliquée. On commence par créer un pointeur qui sera en fait une liste, si le chiffre j est dans la case i alors on ajoute i à l'indice j dans la liste. Ensuite

pour chaque case, on regarde si les chiffres contenu dans la case sont aussi contenu dans d'autres de la sous-grille, si oui on ajoute a une autre liste d'indices cette fois la case j qui contient aussi le sous-set. Si la deuxieme liste contient autant d'indice que la case i contient de candidat dans sa case alors on peut appliquer le hidden-subset. avec la deuxieme liste qu'on a rempli il suffit d'aller aux indices correspondant et de supprimer les autres candidats n'étant pas dans la première case.

### 2.2 Choix et Backtrack

Pour certaines grilles, les heuristiques suffisent, mais pour d'autres, il est nécessaire de faire des choix. On choisit la case contenant le moins de candidat pour faire un choix, puis on prend un candidat aléatoirement parmis ceux présents dans la couleur. Au début on avait choisi de prendre le candidat le plus petit mais il s'avère plus pratique de choisir aléatoirement pour la génération de grilles (on y revient plus tard). Une fois le choix effectué, on copie la grille et on applique le choix a la copie, on appelle alors récursivement notre fonction solver sur la copie ou le choix à été fait. La fonction s'arrête a un moment puisque il y un nombre fini de cases donc un nombre fini de choix, elle nous retourne alors soit une grille résolue soit NULL qui correspond a une grille inconsistante.

Si la fonction retourne dans la copie une grille résolue, si l'utilisateur n'a demandé qu'une seule solution on libère l'espace mémoire occupé par les choix et on retourne la copie résolue pour l'afficher (avec plus de détails si le verbose est activé). Si l'utilisateur a activé le mode 'all', on affiche la grille résolue, on libère l'espace mémoire occupé par les choix et la copie, on enlève le choix effectué auparavant et on applique à nouveau le solver sur la grille, jusqu'a ne plus avoir de choix possibles.

Si la fonction retourne NULL, que ce soit dans le mode 'all' ou non, on libère l'espace mémoire alloué aux choix et a la copie, on retire le choix effectué auparavant puisqu'il est logiquement mauvais et on applique a nouveau le solver à la grille. Au final quand on tombe sur une grille inconsistante, on revient au dernier choix effectué et on en fait un autre.

## 3 Génération

Pour générer un sudoku à résoudre, on commence par générer un sudoku résolu, c'est ici que les choix aléatoires vont nous servir, on génère une grille vide et on la résoud. Après quelques tests, opter pour un choix aléatoire plutôt que la couleur la plus petite ou la plus grande ne fait pas perdre de temps, mais cela permet surtout de générer des grilles aléatoires facilement, sinon le programme ferait toujours les même choix sur une grille vide.

Une fois la grille générée, il va s'agir de retirer des cases jusqu'a avoir $40(+-5)\%$ de cases vides. Je commence par générer aléatoirement une liste d'entier de 0 à $n^2-1$ qui correspondra aux indices de toutes les cases de la grille, en les écrivant sous la forme de $9 * row + col$. On parcourt ensuite celle liste la et c'est la qu'il va y avoir une divergence quant à la génération de grille unique ou non. Si on veut une grille unique, on va créer une copie de la grille en enlevant le chiffre de la case correspondant a l'indice de la liste, on applique ensuite le solver à la copie en mode all et dès que le solver trouve plus d'une solution il s'arrête. En effet si le solver trouve plus d'une solution cela veut dire qu'enlever cette case à la grille ne la rendrait plus unique, on retire alors cette case de la liste et on passe a l'itération suivante. Si il n'est pas important d'avoir une grille avec une seule solution il suffit de supprimer le chiffre de la case dirèctement. Une fois qu'on a enlevé plus de 40% de la grille ou que l'on a parcouru toute la liste ( seulement en mode unique ), on peut rendre la grille générée ou en mode unique si on s'arrête parce que l'on a parcouru toute la grille, on génère une autre grille ainsi qu'une autre liste. Générer une autre liste ne nous assurerait pas d'obtenir une grille unique alors il est plus efficace de générer un autre grille également.

# 4 Optimisation

J'ai bien vérifié a chaque étape de n'avoir aucune perte de mémoire avec valgrind et cela permettait également d'aider a débugger en cas de faute de segmentation, de même gprof m'affiche que les fonctions prenant le plus de temps sont hidden-subset et les bit-hacks, ces deniers ne pouvant être optimisé il ne me reste que hidden-subset a optimiser ou plutôt a moins appeler ainsi que d'ajouter d'autres heuristiques.